

O'REILLY®

第2版



# 像计算机 科学家一样 思考Python

Think Python: How to Think Like a Computer Scientist,  
Second Edition

[美] Allen B. Downey 著  
赵普明 译



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

# 目 录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[O'Reilly Media, Inc.介绍](#)

[前言](#)

[第1章 程序之道](#)

[1.1 什么是程序](#)

[1.2 运行Python](#)

[1.3 第一个程序](#)

[1.4 算术操作符](#)

[1.5 值和类型](#)

[1.6 形式语言和自然语言](#)

[1.7 调试](#)

[1.8 术语表](#)

[1.9 练习](#)

[第2章 变量、表达式和语句](#)

[2.1 赋值语句](#)

[2.2 变量名称](#)

[2.3 表达式和语句](#)

[2.4 脚本模式](#)

[2.5 操作顺序](#)

[2.6 字符串操作](#)

[2.7 注释](#)

[2.8 调试](#)

[2.9 术语表](#)

[2.10 练习](#)

## [第3章 函数](#)

### [3.1 函数调用](#)

### [3.2 数学函数](#)

### [3.3 组合](#)

### [3.4 添加新函数](#)

### [3.5 定义和使用](#)

### [3.6 执行流程](#)

### [3.7 形参和实参\[1\]](#)

### [3.8 变量和形参是局部的](#)

### [3.9 栈图](#)

### [3.10 有返回值函数和无返回值函数](#)

### [3.11 为什么要有函数](#)

### [3.12 调试](#)

### [3.13 术语表](#)

### [3.14 练习](#)

## [第4章 案例研究：接口设计](#)

### [4.1 turtle模块](#)

### [4.2 简单重复](#)

### [4.3 练习](#)

### [4.4 封装](#)

### [4.5 泛化](#)

### [4.6 接口设计](#)

### [4.7 重构](#)

### [4.8 一个开发计划](#)

### [4.9 文档字符串](#)

### [4.10 调试](#)

### [4.11 术语表](#)

### [4.12 练习](#)

## [第5章 条件和递归](#)

### [5.1 向下取整除法操作符和求模操作符](#)

### [5.2 布尔表达式](#)

### [5.3 逻辑操作符](#)

### [5.4 条件执行](#)

### [5.5 选择执行](#)

### [5.6 条件链](#)

[5.7 嵌套条件](#)  
[5.8 递归](#)  
[5.9 递归函数的栈图](#)  
[5.10 无限递归](#)  
[5.11 键盘输入](#)  
[5.12 调试](#)  
[5.13 术语表](#)  
[5.14 练习](#)

## [第6章 有返回值的函数](#)

[6.1 返回值](#)  
[6.2 增量开发](#)  
[6.3 组合](#)  
[6.4 布尔函数](#)  
[6.5 再谈递归](#)  
[6.6 坚持信念](#)  
[6.7 另一个示例](#)  
[6.8 检查类型](#)  
[6.9 调试](#)  
[6.10 术语表](#)  
[6.11 练习](#)

## [第7章 迭代](#)

[7.1 重新赋值](#)  
[7.2 更新变量](#)  
[7.3 while语句](#)  
[7.4 break语句](#)  
[7.5 平方根](#)  
[7.6 算法](#)  
[7.7 调试](#)  
[7.8 术语表](#)  
[7.9 练习](#)

## [第8章 字符串](#)

[8.1 字符串是一个序列](#)  
[8.2 len](#)  
[8.3 使用for循环进行遍历](#)

- [8.4 字符串切片](#)
- [8.5 字符串是不可变的](#)
- [8.6 搜索](#)
- [8.7 循环和计数](#)
- [8.8 字符串方法](#)
- [8.9 操作符in](#)
- [8.10 字符串比较](#)
- [8.11 调试](#)
- [8.12 术语表](#)
- [8.13 练习](#)

## [第9章 案例分析：文字游戏](#)

- [9.1 读取单词列表](#)
- [9.2 练习](#)
- [9.3 搜索](#)
- [9.4 使用下标循环](#)
- [9.5 调试](#)
- [9.6 术语表](#)
- [9.7 练习](#)

## [第10章 列表](#)

- [10.1 列表是一个序列](#)
- [10.2 列表是可变的](#)
- [10.3 遍历一个列表](#)
- [10.4 列表操作](#)
- [10.5 列表切片](#)
- [10.6 列表方法](#)
- [10.7 映射、过滤和化简](#)
- [10.8 删除元素](#)
- [10.9 列表和字符串](#)
- [10.10 对象和值](#)
- [10.11 别名](#)
- [10.12 列表参数](#)
- [10.13 调试](#)
- [10.14 术语表](#)
- [10.15 练习](#)

## [第11章 字典](#)

### [11.1 字典是一种映射](#)

### [11.2 使用字典作为计数器集合](#)

### [11.3 循环和字典](#)

### [11.4 反向查找](#)

### [11.5 字典和列表](#)

### [11.6 备忘](#)

### [11.7 全局变量](#)

### [11.8 调试](#)

### [11.9 术语表](#)

### [11.10 练习](#)

## [第12章 元组](#)

### [12.1 元组是不可变的](#)

### [12.2 元组赋值](#)

### [12.3 作为返回值的元组](#)

### [12.4 可变长参数元组](#)

### [12.5 列表和元组](#)

### [12.6 字典和元组](#)

### [12.7 序列的序列](#)

### [12.8 调试](#)

### [12.9 术语表](#)

### [12.10 练习](#)

## [第13章 案例研究：选择数据结构](#)

### [13.1 单词频率分析](#)

### [13.2 随机数](#)

### [13.3 单词直方图](#)

### [13.4 最常用的单词](#)

### [13.5 可选形参](#)

### [13.6 字典减法](#)

### [13.7 随机单词](#)

### [13.8 马尔可夫分析](#)

### [13.9 数据结构](#)

### [13.10 调试](#)

### [13.11 术语表](#)

### [13.12 练习](#)

## [第14章 文件](#)

### [14.1 持久化](#)

### [14.2 读和写](#)

### [14.3 格式操作符](#)

### [14.4 文件名和路径](#)

### [14.5 捕获异常](#)

### [14.6 数据库](#)

### [14.7 封存](#)

### [14.8 管道](#)

### [14.9 编写模块](#)

### [14.10 调试](#)

### [14.11 术语表](#)

### [14.12 练习](#)

## [第15章 类和对象](#)

### [15.1 用户定义类型](#)

### [15.2 属性](#)

### [15.3 矩形](#)

### [15.4 作为返回值的实例](#)

### [15.5 对象是可变的](#)

### [15.6 复制](#)

### [15.7 调试](#)

### [15.8 术语表](#)

### [15.9 练习](#)

## [第16章 类和函数](#)

### [16.1 时间](#)

### [16.2 纯函数](#)

### [16.3 修改器](#)

### [16.4 原型和计划](#)

### [16.5 调试](#)

### [16.6 术语表](#)

### [16.7 练习](#)

## [第17章 类和方法](#)

### [17.1 面向对象特性](#)

### [17.2 打印对象](#)

[17.3 另一个示例](#)  
[17.4 一个更复杂的示例](#)  
[17.5 init方法](#)  
[17.6 str 方法](#)  
[17.7 操作符重载](#)  
[17.8 基于类型的分发](#)  
[17.9 多态](#)  
[17.10 接口和实现](#)  
[17.11 调试](#)  
[17.12 术语表](#)  
[17.13 练习](#)

[第18章 继承](#)  
[18.1 卡片对象](#)  
[18.2 类属性](#)  
[18.3 对比卡牌](#)  
[18.4 牌组](#)  
[18.5 打印牌组](#)  
[18.6 添加、删除、洗牌和排序](#)  
[18.7 继承](#)  
[18.8 类图](#)  
[18.9 数据封装](#)  
[18.10 调试](#)  
[18.11 术语表](#)  
[18.12 练习](#)

[第19章 Python拾珍](#)  
[19.1 条件表达式](#)  
[19.2 列表理解](#)  
[19.3 生成器表达式](#)  
[19.4 any和all](#)  
[19.5 集合](#)  
[19.6 计数器](#)  
[19.7 defaultdict](#)  
[19.8 命名元组](#)  
[19.9 收集关键词参数](#)  
[19.10 术语表](#)



## [19.11 练习](#)

## [第20章 调试](#)

### [20.1 语法错误](#)

[我一直进行修改，但没有什么区别](#)

### [20.2 运行时错误](#)

#### [20.2.1 我的程序什么都不做](#)

#### [20.2.2 我的程序卡死了](#)

#### [20.2.3 无限循环](#)

#### [20.2.4 无限递归](#)

#### [20.2.5 执行流程](#)

#### [20.2.6 当我运行程序，会得到一个异常](#)

#### [20.2.7 我添加了太多print语句，被输出淹没了](#)

### [20.3 语义错误](#)

#### [20.3.1 我的程序运行不正确](#)

#### [20.3.2 我有一个巨大而复杂的表达式，而它和我预料的不同](#)

#### [20.3.3 我有一个函数，返回值和预期不同](#)

#### [20.3.4 我真的真的卡住了，我需要帮助](#)

#### [20.3.5 不行，我真的需要帮助](#)

## [第21章 算法分析](#)

### [21.1 增长量级](#)

### [21.2 Python基本操作的分析](#)

### [21.3 搜索算法的分析](#)

### [21.4 散列表](#)

### [21.5 术语表](#)

## [译后记](#)

## [译者介绍](#)

## [作者介绍](#)

## [封面介绍](#)

## [欢迎来到异步社区！](#)

## 版权信息

书名：像计算机科学家一样思考Python（第2版）

ISBN：978-7-115-42551-5

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

---

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

---

• 著 [美] Allen B. Downey

译 赵普明

责任编辑 杨海玲

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

- 读者服务热线: (010)81055410

反盗版热线: (010)81055315

## 版权声明

Copyright ©2016 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2016. Authorized translation of the English edition, 2016 O'Reilly Media, Inc., the owner of all rights to publish and sell the same. All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版由O'Reilly Media, Inc.授权人民邮电出版社出版。未经出版者书面许可，对本书的任何部分不得以任何方式复制或抄袭。

版权所有，侵权必究。

## 内容提要

本书以培养读者以计算机科学家一样的思维方式来理解Python语言编程。贯穿全书的主体是如何思考、设计、开发的方法，而具体的编程语言，只是提供了一个具体场景方便介绍的媒介。

全书共21章，详细介绍Python语言编程的方方面面。本书从最基本的编程概念开始讲起，包括语言的语法和语义，而且每个编程概念都有清晰的定义，引领读者循序渐进地学习变量、表达式、语句、函数和数据结构。书中还探讨了如何处理文件和数据库，如何理解对象、方法和面向对象编程，如何使用调试技巧来修正语法错误、运行时错误和语义错误。每一章都配有术语表和练习题，方便读者巩固所学的知识 and 技巧。此外，每一章都抽出一节来讲解如何调试程序。作者针对每章所专注的语言特性，或者相关的开发问题，总结了调试的方方面面。

本书的第2版与第1版相比，做了很多更新，将编程语言从Python 2升级成Python 3，并修改了很多示例和练习，增加了新的章节，更全面地介绍Python语言。

这是一本实用的学习指南，适合没有Python编程经验的程序员阅读，也适合高中或大学的学生、Python爱好者及需要了解编程基础的人阅读。对于第一次接触程序设计的人来说，是一本不可多得的佳作。



## O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

### 业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal



# 前言

## 本书的奇特历史

1999年，我正在为一门Java的编程入门课程备课。这门课我已经教过3个学期，感到有些灰心。课程的不及格率太高，即使是那些及格的学生，也只获得了很低的成就。

我发现问题的之一是教材。它们太厚，有太多冗余的细节，而针对编程技巧的高阶的指导却很不足。而且学生们都有“陷阱效应”的苦恼：开头时很容易，也能循序渐进，但接着在第5章左右，整个地板就突然陷落了。新内容来得太多、太快，以至于我必须花费一学期剩下的全部时间来帮助他们拾回丢失的片段。

开课前两周，我决定自己来编写教材。我的目标有以下几个。

- 尽量简短。学生读10页书，比不读50页书要好。
- 注意词汇。我尝试尽量少用术语，并在第一次使用它们时做好定义。
- 循序渐进。为了避免陷阱效应，我抽出了最困难的课题，并把它们划分成更细的学习步骤。
- 专注于编程，而不是编程语言。我只注意包涵了Java的最小的可用子集，而忽略掉其他。

我需要有一个标题，所以心血来潮选择了“**How to Think Like a Computer Scientist**”。

第 1 版教材很粗糙，但确实有效。学生们读完课本，懂得了足够的基础知识，以至我甚至可以利用课堂时间和他们一起讨论更难、更有趣的话题，并且（最重要的是）可以让学生们有足够的时间在课堂上做练习。

我将这本书按照GNU自由文档许可协议（**GNU Free Documentation License**）发布，让用户可以复制、修改和分发本书。

接下来发生了最酷的事情。**Jeff Elkner**，弗吉尼亚州的一位高中老师，使用了我的书，并且将其翻译成Python语言的版本。他寄给我他的翻译副本，于是我有了一次很奇特的经历——通过读我自己的书来学习Python。通过绿茶出版社（**Green Tea Press**），在2001年我出版了第一个Python版本。

2003年，我开始在欧林学院（**Olin College**）教学，并第一次需要教授Python语言。和Java的对比非常惊人。学生们困扰更少，学会得更多，从事更有意思的项目，总的来说得到了更多的乐趣。

在那之后我一直继续拓展这本书的内容，修改错误，改进示例，并增加新的材料、尤其是练习。

结果就产生了本书，并改用了不那么宏伟堂皇的书名——*Think Python*。部分改动如下所述。

- 我在每章的结尾添加了一节关于调试的说明。这些章节描述寻找和避免bug的通用技巧，并警示Python中容易出错的误区。
- 我增加了更多的练习，小到简短的理解性测试，大到几个实际工程。大部分练习都附带了链接，可以查看我的解答。
- 我添加了一系列案例研究——较长的示例，包括练习、解答以及讨论。
- 我扩展了关于程序开发计划和基础设计模式的讨论。
- 我增加了关于调试和算法分析的章节。

第2版增加了如下几个新特性。

- 全书内容和辅助代码都更新到Python 3。
- 增加了几节，以及更多关于Web的细节，以帮助初学者通过浏览器就能开始运行Python，而不需要过早地面对安装Python的问题。
- 对于第4章的“turtle 模块”，我把实现从以前自己开发的Swampy乌龟绘图包，改为使用更标准的Python模块turtle，它更容易安装，功能也更强大。
- 增加了新的一章“Python拾珍”（第19章），介绍Python提供的一些并不必需，但有时会很方便的特性。

我希望你喜欢这本书，并希望它至少能提供一点帮助，助你学会像计算机科学家那样编程和思考。

——Allen B. Downey

## 本书排版约定

本书使用下列排版约定。

- 中文楷体（英文斜体）：用于新术语、文件名和文件扩展名。
- 黑体字：表示术语表中定义的词汇。
- 等宽字体（**constant width**）：用于程序清单，以及段落中间的代码元素，如变量、函数名、数据库、数据类型、环境变量、语句或关键字等。
- 加粗等宽字体（**constant with bold**）：表示命令或其他应当由用户键入的文本。
- 等宽斜体字（*constant width*）：用于显示需要替换为用户提供的值或由环境确定的值的文本。

## 代码示例的使用

补充材料（代码示例、练习等）可以从  
<http://www.greenteapress.com/thinkpython2/code> 下载。

本书的目的是帮你完成工作。一般来说，只要是本书提供的示例代码，你都可以用在自己的程序和文档中。如果你不是要复制大部分的代码，就不需要联系我们申请授权。例如，写一个程序，里面使用了本书中的几段代码，不需要申请授权。但销售或分发O'Reilly书籍的

示例光盘则需要授权。回答问题中引用本书内容或示例代码，并不需要申请授权，但将本书中大量的代码引入你的产品文档则需要授权。

在引用本书内容时，我们并不强求但鼓励你注明出处。引用通常包括书名、作者、出版社和ISBN。例如：“*Think Python, 2nd Edition* by Allen B. Downey (O’Reilly). Copyright 2016 Allen Downey, 978-1-4919-3936-9”。

如果你觉得自己对本书代码示例的使用超出了上述授权范围，可以随时联系我们：[permissions@oreilly.com](mailto:permissions@oreilly.com)。

## 联系我们

请将关于本书的评论和问题发给出版商。

美国：

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）

奥莱利技术咨询（北京）有限公司

我们为本书提供了专门的网页，上面有勘误表、示例，以及其他额外的信息，可以通过[http://bit.ly/think-python\\_2E](http://bit.ly/think-python_2E)访问该网页。

如果想对本书进行评论或想问技术问题，请将邮件发到 [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)。

想了解更多关于我们的书籍、课程、会议，以及新闻等信息，请登录我们的网站：<http://www.oreilly.com>。

我们的其他联系方式如下。

Facebook: <http://facebook.com/oreilly>

Twitter: <http://twitter.com/oreillymedia>

YouTube: <http://www.youtube.com/oreillymedia>

## 致谢

非常感谢Jeff Elkner，他将我的Java书翻译成Python，这使我我开始了这个项目，并向我介绍了Python语言，结果Python成为我最喜爱的编程语言。

还要感谢Chris Meyers，他在*How to Think Like a Computer Scientist*一书中贡献了好几节。

感谢自由软件基金会（Free Software Foundation）开发了GNU自由文档协议，让我和Jeff以及Chris的合作成为可能。感谢创用CC

(Creative Commons) 开发了我们现在使用的协议。

感谢Lulu，负责*How to Think Like a Computer Scientist*的编辑。

感谢O'Reilly Media负责Think Python一书的编辑。

感谢所有参与了本书早期版本编写的学生，以及所有（下面列出的）贡献者提供的修订和建议。

## 贡献者列表

在最近几年中，超过100名眼光犀利、思维敏捷的读者给我寄来了建议和修订。他们对这个项目的贡献和热情，对我是极大的帮助。

如果你有建议或者修订意见，请发邮件到 [feedback@thinkpython.com](mailto:feedback@thinkpython.com)。如果我根据你的回馈做出了修改，会将你加入贡献者列表中（除非你要求被隐藏）。

如果你给出错误出现的位置的部分语句，会让我更容易搜索。页码或者章节编号也可以，但并不那么容易处理。谢谢！

- Lloyd Hugh Allen对8.4节提出了修订建议。
- Yvon Boulianne对第5章提出了一个语义错误的修订建议。
- Fred Bremmer对2.1节提出了一个修订建议。
- Jonah Cohen编写了Perl脚本将本书的LaTeX源码转换成美丽的HTML。
- Michael Conlon提出了第2章的一个语法错误，并提出第1章的格式改进，并且他开启了对解释器的技术讨论。

- Benoit Girard寄来一个对5.6节的有趣的修订。
- Courtney Gleason 和 Katherine Smith 编写了`horsebet.py`，在本书的早期版本中作为一个案例研究。他们的程序现在可以在网站上找到。
- Lee Harr提交了很多修订建议，我们没有空间在这里一一列出，并且他确实应当被列为本书的一位主要编辑。
- James Kaylin是一名使用本书的学生。他提交了许多修订。
- David Kershaw 修正了3.10节中错误的`catTwice` 函数。
- Eddie Lam提出了第1章、第2章和第3章的很多修订建议，他也修正了`Makefile`，这样第一次运行时会自动建立索引。他也帮助我们设置了一个版本管理方案。
- Man-Yong Lee 寄来了对2.4节中的示例代码的修订。
- David Mayo指出第1章中的单词“unconsciously”需要被修改为“subconsciously”。
- Chris McAloon 寄来了对3.9节和3.10节的一些修订。
- Matthew J. Moelter是本书的长期贡献者，提出了很多修订建议。
- Simon Dicon Montford报告了第3章中缺失的函数定义以及几个错别字。他也发现了第13章中的`increment` 函数的错误。
- John Ouzts 修正了第3章中“返回值”的定义。
- Kevin Parks对关于本书如何分布提出了有价值的评论和建议。
- David Pool 发来了第1章中术语表中的错别字，以及鼓励的赞美之言。
- Michael Schmitt寄来了关于文件和异常的章节的修订建议。
- Robin Shaw指出了13.1节中的一个错误，`printTime`函数在一个示例中没有定义就使用了。



- Paul Sleigh在第7章中找到一个错误，并发现了Jonah Cohen用于生成HTML的Perl脚本的bug。
- Craig T. Snydal在德鲁大学（Drew University）的一门课上试验这个课本，他提出了好几个有价值的建议和修订。
- Ian Thomas和他的学生们使用这本书作为编程课程的教材。他们第一个尝试使用本书后半部分的章节，并且提出了许多勘误和建议。
- Keith Verheyden发来了第3章的一个修正。
- Peter Winstanley让我们知道了第3章的拉丁文中一个长期存在的错误。
- Chris Wrobel修正了文件I/O和异常一章的代码错误。
- Moshe Zadka对本书有不可估量的贡献。他编写了关于字典的一章的第1版草稿，并在本书的早期阶段持续提供指导。
- Christoph Zwerschke发来了几个勘误和教学法的建议，并解释了gleich和selbe的区别。
- James Mayer发送给我们非常多的拼写错误，包括贡献者列表中的两个错误。
- Hayden McAfee发现了两个示例之间潜在的冲突。
- Angel Arnal是翻译本书的西班牙语版本的国际团队的一员。他也发现了英文版中的几个错误。
- Tauhidul Hoque 和Lex Berezchny创建了第1章中的图表，并改进了很多其他图表。
- Dr. Michele Alzetta发现了第8章中的一个错误，并发来了一些有趣的教学法评论，以及关于斐波那契数列和Old Maid的建议。

- Andy Mitchell发现了第1章中的一个录入错误，以及第2章中一个错误的示例。
- Kalin Harvey对第7章的一个说明提供了建议，并发现了几个录入错误。
- Christopher P. Smith发现了几个录入错误，并帮助我们更新本书到Python 2.2。
- David Hutchins发现了前言中的一个错别字。
- Gregor Lingl在奥地利维也纳的一个高中教授Python。他正在翻译本书的德文版，并发现了第5章中的几个错误。
- Julie Peters发现了前言中的一个错别字。
- Florin Oprina发来一个makeTime 的改进，printTime 的一个修正，以及发现的一个重要的录入错误。
- D. J. Webre对第3章的一个说明提出了建议。
- Ken在第8、9、11章中发现了好几个错误。
- Ivo Wever在第5章发现一个录入错误，并对第3章中的一个说明提出了建议。
- Curtis Yanko对第2章中的一个描述提出了建议。
- Ben Logan发来许多发现的录入错误，并发现了翻译HTML的问题。
- Jason Armstrong发现了第2章中一个漏掉的词。
- Louis Cordier发现了第16章中有一个代码和文本不一致的地方。
- Brian Cain在第2章和第3章中提出了几个描述的改进建议。
- Rob Black发来了许多勘误，包括一些针对Python 2.2的修改。
- 巴黎中央理工大学的Jean-Philippe Rey发来了一些补丁，包括对Python 2.2的更新，以及其他一些细心的改进。

- 乔治华盛顿大学的Jason Mader提供了许多有用的建议和改正。
- Jan Gundtofte-Bruun提醒我们“a error”应改为“an error”。
- Abel David和Alexis Dinno 提醒我们“matrix”的复数形式是“matrices”而不是“matrixes”。这个错误在书中已经存在了多年，但两个姓名以同样的字母开头的读者同一天报告了它。真的很奇怪。
- Charles Thayer鼓励我们删除掉一些语句结尾的分号，并建议我们理清“形参”和“实参”的使用。
- Roger Sperberg指出了第3章的一个逻辑错误。
- Sam Bull指出了第2章中一段令人困惑的描述。
- Andrew Cheung指出了两处“定义前先使用”的错误。
- C.Corey Capel发现缺了单词，以及第4章的一个录入错误。
- Alessandra 帮助我们理清了一些关于Turtle的困惑。
- Wim Champagne在字典示例中发现一个错误。
- Douglas Wright在弧度计算中发现了一个除法向下取整的错误。
- Jared Spindor发现了一处句尾的无用词。
- Lin Peiheng发来了许多很有用的建议。
- Ray Hagtvedt发来了两处错误和一处不是那么错的错误。
- Torsten Hübsch指出Sawmpy中的一处不一致。
- Inga Petuhhov修正了第14章中的一个示例。
- Arne Babenhauserheide发来了几个有用的勘误。
- Mark E. Casida非常善于发现重复的单词。
- Scott Tyler填上了一个缺失的“that”，并发来了一堆勘误。
- Gordon Shephard发来了几个勘误，每个都用单独的邮件。
- Andrew Turner发现了第8章中的一个错误。

- Adam Hobart修正了一个在弧度计算中除法向下取整的错误。
- Daryl Hammond和Sarah Zimmerman指出我过早提出了`math.pi`。  
。并且Zim发现了一个录入错误。
- George Sass在调试章节中发现了一个bug。
- Brian Bingham建议了练习11-10。
- Leah Engelbert-Fenton指出我用`tuple`作为变量名称，这恰恰违反了我自己的建议。然后他发现了一堆录入错误以及一个“定义前先使用”。
- Joe Funke发现了一个录入错误。
- Chao-chao Chen在斐波那契示例中发现了一个不一致处。
- Jeff Paine知道space和spam的区别。
- Lubos Pintes发来一个录入错误。
- Gregg Lind和Abigail Heithoff建议了练习14-4。
- Max Hailperin发来了许多勘误和建议。Max是非凡的*Concrete Abstractions*（Course Technology, 1998）一书的作者之一。在读完本书之后你可能会想要读那本书。
- Chotipat Pornavalai在一个错误信息中发现了一个错误。
- Stanislaw Antol寄来了一个很有用的建议列表。
- Eric Pashman对第4章到第11章发来了许多勘误。
- Miguel Azevedo发现了一些录入错误。
- Jianhua Liu发来了一长列勘误。
- Nick King发现了一个缺失单词。
- Martin Zuther发来了一长列建议。
- Adam Zimmerman发现了我举例的一个“实例”中的不一致处，以及其他一些错误。

- Ratnakar Tiwari建议加一个脚注说明什么是“退化”三角形。
- Anurag Goel提出了`is_abecedarian`的另一个解答，并发来其他一些勘误。他还知道如何拼写Jane Austen。
- Kelli Kratzer发现了一个录入错误。
- Mark Griffiths指出了第3章中的一个令人困惑的示例。
- Roydan Ongie发现了我的牛顿方法的一个错误。
- Patryk Wolowiec帮我解决了一个HTML版本的问题。
- Mark Chonofsky告诉我Python 3中的新关键字。
- Russell Coleman帮我修正了几何错误。
- Wei Huang发现了几处录入错误。
- Karen Barber发现了本书中最古老的录入错误。
- Nam Nguyen发现了一个录入错误，并指出我使用了装饰器模式但没有用它的名字。
- Stéphane Morin发来了一些建议和勘误。
- Paul Stoop修改了一个`uses_only`中的录入错误。
- Eric Bronner指出了关于操作符顺序的讨论中的一个困惑之处。
- Alexandros Gezerlis提交的建议的数量和质量都设置了一个新的标准。我们非常感谢他！
- Gray Thomas知道哪边是左哪边是右。
- Giovanni Escobar Sosa发来一长列的勘误和建议。
- Alix Etienne修正了一个URL。
- Kuang He发现一个录入错误。
- Daniel Neilson修正了一个关于操作符顺序的错误。
- Will McGinnis指出`polyline`在两个地方定义的不同。
- Swarup Sahoo发现了一个缺失的分号。

- Frank Hecker指出一个练习不细致，并发现了几个坏链接。
- Animesh B帮助我清理了一个令人困惑的示例。
- Martin Caspersen发现了两处取整错误。
- Gregor Ulm发来一些勘误和建议。
- Dimitrios Tsirigkas建议我更清晰地描述一个练习。
- Carlos Tafur发送了一整页勘误和建议。
- Martin Nordsletten在一个练习解答中找到了一个bug。
- Lars O. D. Christensen找到了一个失效的引用。
- Victor Simeone 找到了一个录入错误。
- Sven Hoexter指出一个叫作的变量名覆盖了内置函数名。
- Viet Le找到了一个录入错误。
- Stephen Gregory指出Python 3中cmp 的问题。
- Matthew Shultz告知我一个失效链接。
- Lokesh Kumar Makani告知我几个失效链接，以及出错消息的改变。
- Ishwar Bhat修正了我对费马大定理的描述。
- Brian McGhie建议了一个更清晰的阐述。
- Andrea Zanella将本书翻译成意大利语，并发送了一些勘误。
- 非常感谢Melissa Lewis和Luciano Ramalho出色的评论，以及对本书第2版的建议。
- 感谢PythonAnywhere的Harry Percival帮助人们在浏览器中运行Python。
- Xavier Van Aubel对第2版做出了几个有用的修正。

# 第1章 程序之道

本书的目标是教会你像计算机科学家一样思考。这种思考方式综合了数学、工程学以及自然科学的一些最优秀的特性。计算机科学家与数学家类似，他们使用形式语言来描述理念（特别是计算）；与工程师类似，他们设计产品，将元件组装成系统，对不同的方案进行评估选择；与自然科学家类似，他们观察复杂系统的行为，构建科学假说，并检验其预测。

作为计算机科学家，最重要的技能就是**问题求解**。问题求解是发现问题、创造性地思考解决方案以及清晰准确地表达解决方案的能力。实践证明，学习编程的过程，正是训练问题求解能力的绝佳机会。这也是本章标题用“程序之道”的原因。

一方面，你将学会编程，其本身就是一个非常有用的技能；另一方面，你可以使用编程作为工具，去达到更高的目标。随着本书的深入，那个目标会逐渐明晰。

## 1.1 什么是程序

**程序**是指一组定义如何进行计算的指令的集合。这种计算可能是数学计算，如解方程组或者查找多项式的根，也可以是符号运算，如搜索和替换文档中的文本，或者图形相关的操作，如处理图像或播放视频。

在不同的编程语言中，程序的细节有所不同，但几乎所有编程语言中都会出现以下几类基本指令。

- 输入：从键盘、文件或者其他设备中获取数据。
- 输出：将数据显示到屏幕，保存到文件中，或者发送到网络上等。
- 数学：进行基本数学操作，如加法或乘法。
- 条件执行：检查某种条件的状态，并执行相应的代码。
- 重复：重复执行某种动作，往往在重复中有一些变化。

信不信由你，这差不多就是全部了。你所遇到过的所有程序，无论多么复杂，都是由类似上面的这些指令组成的。所以我们可以把编程看作一个将大而复杂的任务分解为更小的子任务的过程，不断分解，直到任务简单到足以由上面的这些基本指令组合完成。

## 1.2 运行Python

Python入门的挑战之一在于你可能需要自己在电脑上安装Python及相关软件。如果你熟悉自己的操作系统，而且习惯于命令行界面，那么安装Python不是什么问题。但对于初学者来说，同时学习编程和系统管理命令两件事，有时候是非常痛苦的。

为了避免这个问题，我推荐你开始先在浏览器中运行Python，等熟悉了Python语言之后，我再向你介绍如何在电脑上安装Python。

用于运行Python的网站有不少。如果你已经找到一个喜欢的，就可以直接去用。如果没有，我推荐PythonAnywhere。我在



<http://tinyurl.com/thinkpython2e>上提供了详细的入门指导。

有两个版本的Python，分别为Python 2和Python 3。它们很类似，所以如果你学会了一个版本，也能很容易地切换到另一个版本。实际上，作为初学者，你会遇到的两者之间的区别非常少。本书是针对Python 3编写的，但我也会给出一些关于Python 2的注意事项。

Python**解释器**是一个读取并执行Python代码的程序。根据所在环境的不同，你可能需要点击程序图标，或者在命令行中键入python命令来启动解释器。当它启动以后，可以看到如下输出：

```
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

前3行文本包含了解释器和所运行的操作系统的信息，所以可能与你看到的有些区别。但你应当检查版本号是否以3开头（本例所示的是3.4.0），表示你使用的是Python 3的解释器。如果版本号以2开头，那么（你肯定猜到了）解释器是Python 2。

最后一行是一个**提示符**，表明解释器已经准备好，等待你键入代码。如果你键入一行代码并按下Enter键，解释器会显示结果：

```
>>> 1 + 1
2
```

## 1.3 第一个程序

依照传统，用新语言编写的第一个程序叫“Hello, World!”，因为这个程序所做的事情就是只显示“Hello, World!”。在Python中，它是这个样子：

```
>>> print('Hello, World!')
```

这是 **print 语句** 的一个示例。**print** 并不会真往纸上打印文字，而是在屏幕上显示结果。在这个例子中，输出的结果是：

```
Hello, World!
```

程序中的引号表示要显示的文本的开始和结束，在输出结果中它们并不显示。

括号表示**print** 是一个函数。我们将在第3章中讨论函数。

在Python 2中，**print** 语句略有不同。它不是一个函数，所以不使用括号：

```
>>> print 'Hello, World!'
```

这个区别的意义在后面会慢慢显现，但现在只需要知晓就足够了。

## 1.4 算术操作符

介绍完“Hello, World”之后，接下来是算术操作。Python提供了**操作符**，即像加号或减号这样的用来表达计算操作的特殊符号。

操作符+、-和\*分别表示进行加法、减法和乘法运算，如下面示例所示：

```
>>> 40 + 2
42
>>> 43 - 1
42
>>> 6 * 7
42
```

操作符/表示除法运算：

```
>>> 84 / 2
42.0
```

这里你可能会奇怪为什么结果是**42.0**而不是**42**。我会在下一节解释。

最后，操作符\*\*表示进行指数运算。也就是说，会把一个数按指数进行乘方：

```
>>> 6**2 + 6
42
```

在其他一些语言中，指数操作用^符号表示，但在Python中^这个符号已经用来表示二进制按位运算XOR了。如果你不熟悉按位运算，结果可能会让你感到奇怪：

```
>>> 6 ^ 2
4
```

本书我不会讨论按位操作符，但读者可以在 <http://wiki.python.org/moin/BitwiseOperator> 上阅读相关文档。

## 1.5 值和类型

**值**（value）是程序操作的最基本的东西，如一个字母或者数字。前面我们见过一些值，如 `2`、`42.0` 以及 `'Hello, World!'`。

这些值属于不同的**类型**（type）：`2` 是**整型**（integer）的，`42.0` 是**浮点型**（floating-point）的，而 `'Hello, World!'` 是**字符串**（string）类型的，这么称呼是因为它是由一堆字母“串连”起来的。

如果不确认一个值的类型，解释器可以告诉你：

```
>>> type(2)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('Hello, World!')
<type 'str'>
```

在这些结果中，单词“class”（类）被用于某一类型中，这是一种值类型。

不足为奇，整数属于 `'int'` 类型，字符串属于 `'str'` 类型，而浮点数属于 `'float'` 类型。

那么 '2' 和 '42.0' 这样的值呢？它们看起来像是数字，但又使用字符串常用的引号括起来：

```
>>> type('2')
<type 'str'>
>>> type('42.0')
<type 'str'>
```

它们是字符串。

当输入一个很大的数字时，你可能会忍不住想在数字中间加上逗号，就像 1,000,000 这样。在 Python 中这并不是合法的整数，但它凑巧又是一个合法的表达式：

```
>>> 1,000,000
(1, 0, 0)
```

当然，这和我们预期的完全不同！Python 把 1,000,000 解释成一个用逗号分隔的整数序列。关于这种序列在本书后面可以学到更多内容。

## 1.6 形式语言和自然语言

**自然语言** 是指人们所说的语言，如英语、西班牙语和法语。它们不是由人设计而来的（虽然人们会尝试加以语法限制），而是自然演化而来的。

**形式语言** 则是人们为了特殊用途设计的语言。例如，数学上使用的符号体系是一种特别擅于表示数字和符号之间关系的形式语言；化

学家则使用另一种形式语言来表示分子的化学结构。而最重要的是：

**编程语言是人们为了表达计算过程而设计出来的形式语言。**

形式语言倾向于对**语法**做出严格的限制。例如， $3 + 3 = 6$ 是语法正确的数学表达式，但 $3+ = 3\$6$ 则不是。 $\text{H}_2\text{O}$ 是语法正确的化学方程式，而 $_2\text{Zz}$ 则不是。

语法规则有两种，分别适用于**记号**（token）和结构（structure）。记号是语言的基本元素，如词、数字和化学元素。 $3+ = 3\$6$ 的一个问题就是\$在数学表达式中（至少就我所知）不是合法记号。相似地， $_2\text{Zz}$ 不合法是因为并不存在缩写为Zz的化学元素。

第二种语法规则指定记号所组合的方式。数学等式 $3+ = 3$ 不合法，因为虽然+和=是合法记号，但不能将它们连续放置。相似地，在化学表达式里，下标数字应该出现在元素名称之后，而不是之前。

“This is @ well-structured Engli\$h sentence with invalid t\*kens in it.”是一个结构良好，但包含非法记号的英语语句。“This sentence all valid tokens has, but invalid structure with.”这句话所有的记号都合法，但是语句结构不合法。

当你阅读英语的句子或形式语言的语句时，需要弄清句子的结构是什么（虽然在自然语言中这个过程是下意识完成的）。这个过程称为**语法分析**。

虽然形式语言和自然语言有很多共同的特点——记号、结构、语法以及语义，但它们也有一些区别。

- 歧义性：自然语言充满了歧义，人们通过上下文线索和其他信息来处理这些歧义。形式语言通常设计为几乎或者完全没有歧义，即不论上下文环境如何，任何表达式都只有一个含义。
- 冗余性：为了弥补歧义，减少误解，自然语言采用大量的冗余。因此，自然语言往往很啰嗦。形式语言则相对不那么冗余，更加简洁。
- 字面性：自然语言充满了习惯用语和比喻。例如，有人说，“硬币掉了”（The penny dropped [\[1\]](#)），并不一定是硬币，也不一定是有什么掉了。形式语言则严格按照它的字面意思表达含义。

因为我们都说着自然语言长大，有时候很难适应形式语言。在某种意义上，形式语言和自然语言的区别与诗词和散文的区别类似，而且程度更甚。

- 诗词：字词的使用，既考虑它们的音韵，也考虑到它们的意义，而整首诗合起来表达某种意境或情绪反应。歧义不仅常见，而且常常是刻意为之。
- 散文：字词的意义更加重要，而且句子的结构也提供更多的意义。散文比诗词更容易分析，但仍然有不少歧义。
- 程序：计算机程序的意义不含歧义，直接如字面所指。完全可以通过它的记号和结构理解其意义。

形式语言的密度远远大于自然语言，所以阅读起来需要花费更多的时间。还有，结构非常重要，所以直接自顶向下、从左至右的阅读顺序并不一定是最好的。相反，要试着学会在头脑中解析程序，辨别出记号并解析出结构。最后，细节很重要。在自然语言中常常可以忽

略的小错误，如拼写错误或者标点符号错误，在形式语言中往往会造成很大的差别。

## 1.7 调试

程序是很容易出错的。因为某种古怪的原因，程序错误被称为 **bug**，而查捕bug的过程称为**调试**（debugging）。

一个程序中可能出现3种类型的错误：语法错误、运行时错误和语义错误。对它们加以区分，可以更快地找到错误。

编程，特别是调试，有时候会引发强烈的情绪。如果你挣扎于一个困难的bug，可能会感觉到愤怒、沮丧以及窘迫。

有证据表明，人们会像对待人一样对待电脑。当电脑良好完成工作时，我们会把它们当作队友，而当它们难以控制、粗暴无礼的时候，我们会按照对待那些粗暴固执的人一样对待它们（*The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*，Reeves和Nass著）。

对这些反应行为有所准备，可能会帮助你更好地对待电脑。一种方法是把它当作你的雇员，它有一定的长处，如速度和精度，也有特定的弱点，如没有同情心和无法顾全大局。

你的任务是做一个好经理：设法扬长避短，并找到方法控制你的情绪去面对问题，而不是让你的反应影响工作效率。



学习调试可能会带来挫折感，但它是一个有价值的技能，并在编程之外还有很多用途。每章的结尾处都有一节类似于本节的关于调试技巧的讨论。希望它们能带来帮助！

## 1.8 术语表

问题求解（problem solving）：总结问题、寻找解决方案以及表达解决方案的过程。

高级语言（high-level language）：设计来方便人们读写的编程语言，如Python。

低级语言（low-level language）：设计来方便计算机执行的编程语言，也被称为“机器语言”或“汇编语言”。

可移植性（portability）：程序的一种属性：可以在多种类型的计算机上运行。

解释器（interpreter）：一个读取其他程序并执行其内容的程序。

提示符（prompt）：解释器显示的文字，提示用户已经准备好接收用户的输入。

程序（program）：一系列代码指令的集合，指定一种运算。

print语句（print statement）：一个指令，可以通知Python解释器在屏幕上显示一个值。

操作符（**operator**）：一种特殊符号，用来表达加法、乘法或字符串拼接等简单运算。

值（**value**）：程序操作的数据基本单位，如一个数字或一个字符串。

类型（**type**）：值的类别。到目前为止我们已经见过的类型有整数（**int**）、浮点数（**float**）和字符串（**str**）。

整型（**integer**）：用来表示整数的类型。

浮点型（**floating-point**）：用来表示带小数部分的数的类型。

字符串（**string**）：用来表示一串字符的类型。

自然语言（**natural language**）：自然演化而来的人们所说的语言。

形式语言（**formal language**）：人们设计为某些特定目的（如表达数学概念或者计算机程序）设计的任何一种语言。所有编程语言都属于形式语言。

记号（**token**）：程序的语法结构的最基本单位，类似于自然语言中的词。

语法（**syntax**）：用于控制程序结构的规则。

语法分析（**parse**）：检查程序并分析其语法结构。

bug: 程序中的错误。

调试 (debugging): 发现和纠正bug的过程。

## 1.9 练习

### 练习 1-1

在计算机前阅读本书是一个好主意，因为你可以边看边试验书中的示例。

每当你试验新的语言特性时，应当试着故意犯错。例如，在“Hello, World!”程序中，如果少写一个引号，会发生什么？如果两个引号都不写，会怎么样？如果把`print`拼写错了，会如何？

这种试验会帮你记住所读的内容，也能帮你学会调试，因为这样能看到不同的出错消息代表着什么。现在故意犯错总比今后在编码中意外出错好。

1. 在`print`语句中，如果漏掉一个括号，或者两个都漏掉，会发生什么？

2. 如果正尝试打印一个字符串，那么若漏掉一个或所有的引号，会发生什么？

3. 可以使用一个负号来表示负数，如`-2`。如果在数字之前放一个正号，会发生什么？如果是`2++2`呢？

4. 在数学标记里，前置0是没有问题的，如02。在Python中也这么做会发生什么？

5. 如果在两个值之间不放任何操作符，会发生什么？

## 练习1-2

启动Python解释器，把它当作计算器使用。

1. 在42分42秒中，一共有多少秒？

2. 10千米相当于多少英里？提示：1英里相当于1.61千米。

3. 如果你用42分42秒跑完10千米，那么你的平均速度（跑1千米需要的分钟和秒数）是多少？平均速度是多少千米每小时？

---

[1] “The penny dropped”在英语里的意思是：经过一段困惑后，突然理解了某个事情。——译者注

## 第2章 变量、表达式和语句

编程语言最强大的特性之一是操纵**变量**的能力。变量是指向一个值的名称。

### 2.1 赋值语句

**赋值语句**可以建立新的变量，并给它们赋值：

```
>>> message = 'And now for somthing completely different'
>>> n = 17
>>> pi = 3.1415926535897932
```

这个例子有3个赋值。第一个将一个字符串赋给叫作**message**的变量；第二个将**17**赋值给**n**；第三个将 $\pi$ 的（近似）值赋给变量**pi**。

在纸上表达变量的一个常见方式是写下名称，并用箭头指向其值。这种图称为**状态图**，因为它显示了每个变量所在的状态（请将它看作变量的心理状态）。图2-1显示了前面例子的状态图。

```
message —> 'And now for something completely different'
      n —> 17
      pi —> 3.1415926535897932
```

图2-1 状态图

## 2.2 变量名称

程序员常常选择有意义的名称作为变量名——以此标记变量的用途。

变量名可以任意长短。它可以包含字母和数字，但必须以一个字母开头。使用大写字母是合法的，但变量名使用小写字母开头是个好主意（后面你会看到为何如此）。

下划线“\_”可以出现在变量名称中。它经常出现在由多个词组成的变量名中，如`your_name` 或 `airspeed_of_unladen_swallow`。

如果给变量取非法的名称，会得到一个语法错误：

```
>>> 76trombones = 'big parade'
SyntaxError: invalid syntax
>>> more@ = 1000000
SyntaxError: invalid syntax
>>> class = 'Advanced Theoretical Zymurgy'
SyntaxError: invalid syntax
```

`76trombones` 非法，因为它以数字开头。`more@` 非法，是因为它包含了一个非法字符`@`。但`class` 有什么问题？

原因是`class` 是Python的一个**关键字**。解释器通过关键字来识别程序的结构，并且它们不能用来作为变量名称。

Python 2共有31个关键字：

<code>False</code>	<code>class</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>None</code>	<code>continue</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>True</code>	<code>def</code>	<code>from</code>	<code>nonlocal</code>	<code>while</code>

and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

你并不需要记住这个清单。在大多数开发环境中，关键字会以不同的颜色显示。如果把它们当作变量来用，会很容易发现。

## 2.3 表达式和语句

**表达式** 是值、变量和操作符的组合。单独一个值也被看作一个表达式，单独的变量也是如此。所以下面都是合法的表达式：

```
>>> 42
42
>>> n
17
>>> n + 25
42
```

当你在提示符之后键入一个表达式时，解释器会对其进行**求值**，即尝试找到该表达式的最终值。在本例中，变量`n`的值是17，而表达式`n + 25`的值是42。

**语句** 是一段会产生效果的代码单元，如创建新变量或者显示一个值。

```
>>> n = 17
>>> print(n)
```

第一行是一个赋值语句，将值**17** 赋给变量**n**。第二行是一个 **print** 语句，显示变量**n** 的值。

当键入一行语句之后，解释器会执行它，也就是说会按照语句所说的来做。通常来说，语句本身没有值。

## 2.4 脚本模式

到目前为止我们都是在**交互模式**（**interactive mode**）下运行 **Python**，直接与解释器打交道。交互模式非常适合入门，但是，如果你需要编写超过几行的代码，它可能显得有点儿笨拙。

另一种编程模式是把代码保存称为**脚本** 的文件中，并以**脚本模式**（**script mode**）运行解释器，执行脚本。依照惯例，**Python**脚本文件通常以 **.py** 结尾。

如果你已经了解在自己的电脑上如何创建和运行脚本，就可以继续学习了。否则我再次建议使用**PythonAnywhere**。我在 <http://tinyurl.com/thinkpython2e> 上写下了如何在脚本模式下运行的指导。

由于**Python**提供了两种运行模式，你可以在交互模式中尝试代码片段，然后将其放到脚本中。但交互模式和脚本模式还是有一些区别的，可能会引起困惑。

例如，如果使用**Python**作为计算器，你可能会输入：



```
>>> miles = 26.2
>>> miles * 1.61
42.182
```

第一行给变量**miles** 赋值，但没有可见的效果。第二行是一个表达式，所以解释器对其进行求值，并显示结果。于是我们知道马拉松的长度大概是42千米。

但如果将上面同样的代码写入到脚本中并运行，则得不到任何输出。在脚本模式中，一个单独的表达式，也是没有可见效果的。Python 实际上会对表达式进行求值，但不会显示其结果。除非你叫它这么做：

```
miles = 26.2
print (miles * 1.61)
```

这种现象一开始可能会让人迷惑。

脚本通常包含一系列的语句。如果语句超过一行，那么会随着语句执行的顺序一行行显示结果。

例如，脚本

```
print(1)
x = 2
print(x)
```

产生如下结果

```
1
2
```

赋值语句不会产生任何输出。

为了验证你的理解，可以在Python解释器中输入下面的语句，看它们做了什么：

```
5
x = 5
x + 1
```

现在把同样的语句存入到一个脚本文件并运行。输出是什么？修改脚本，将所有的表达式都转换成`print` 语句，再运行一遍。

## 2.5 操作顺序

当一个表达式中出现多个操作符时，求值的顺序依赖于**优先级规则**。对数学操作符，Python遵守数学的传统规则。缩略词**PEMDAS**可以帮助记忆这些规则：

- 括号（**P**，**P**arentheses）拥有最高的优先级，并可以用来强制表达式按照你需要的顺序进行求值。因为括号中的表达式会先执行，所以`2*(3-1)`的结果是4，而`(1+1)**(5-2)`的结果是8。你也可以利用括号使得表达式更加易读，就像`(minute*100)/60`这样，即使这里增加括号并不会改变结果。
- 乘方（**E**，**E**xponentiation）操作拥有次高的优先级，所以`1+2**3`的结果是9，而不是27，而且`2 * 3**2`的结果是18，而不是36。
- 乘法（**M**，**M**ultiplication）和除法（**D**，**D**ivision）优先级相同，并且高于亦有相同优先级的加法（**A**，**A**ddition）和减法（**S**，

Substraction) 。所以 $2*3-1$  是5，而不是4，并且 $6+4/2$  是8，而不是5。

- 优先级相同的操作按照自左向右的顺序求值（除了乘方以外）。所以表达式 $\text{degrees}/2*\text{pi}$ ，除法在乘法之前执行，结果乘以 $\text{pi}$ 。如果想除以 $2\pi$ ，可以使用括号，或者写为 $\text{degrees}/2/\text{pi}$ 。

其他操作符的优先级，我并不会花太多功夫记下来。如果只看表达式不能确定的话，使用括号指明优先级即可。

## 2.6 字符串操作

通常来说，字符串不能进行数学操作。即使看起来像数字也不行。下面的操作是非法的：

```
'2' - '1'      'eggs'/'easy'      'third'*'a charm'
```

但有两个例外：+和\*。

操作符+进行字符串**拼接**（string concatenation）操作，意即将前后两个字符首尾连接起来。例如：

```
>>> first = 'throat'
>>> second = 'warbler'
>>> first + second
throatwarbler
```

操作符\*也适用于字符串；它进行重复操作。例如， $\text{'Spam'}*3$  的结果是 $\text{'SpamSpamSpam'}$ 。如果\*的两个操作对象之一是字符串，那

另一个必须是整数。’

字符串的+和\*的应用，实际上和数字的加法与乘法类似。就像 $4*3$ 与 $4+4+4$ 相等一样，我们预期 `'Spam'*3` 与 `'Spam'+'Spam'+'Spam'` 也相等，实际也确实如此。另一方面，字符串的拼接与重复操作和整数的加法与乘法操作也有很大的不同。你能够想出加法的一个属性，字符串拼接操作并不支持吗？

## 2.7 注释

当程序变得更大更复杂时，读起来也更困难。形式语言很紧凑，经常会遇到一段代码，却很难弄清它在做什么、为什么那么做。

因此，在程序中加入自然语言的笔记来解释程序在做什么，是个好主意。这种笔记被称为**注释**（comments），它们以# 开头：

```
# compute the percentage of the hour that has elapsed
percentage = (minute * 100) / 60
```

在这个例子里，注释单独占据一行。也可以把注释放到代码行的结尾：

```
percentage = (minute * 100) / 60    # percentage of an hour
```

从#开始到行尾的注释内容都会被解释器忽略掉——它们对程序本身运行没有任何影响。

注释最重要的用途在于解释代码并不显而易见的特性。我们可以合理地认为读者可以看懂代码在做什么，因此使用注释来解释为什么这么做，要有用得多。

下面这段注释与代码重复，毫无用处：

```
v = 5          # 将5赋值给v
```

而下面这段注释则包含了代码中看不到的有用信息：

```
v = 5          # 速度，单位是米/秒
```

选择好的变量名称，可以减少注释的需要，但长名字也会让复杂表达式更难阅读，所以这两者之间需要衡量取舍。

## 2.8 调试

一个程序中可能出现3种错误：语法错误、运行时错误和语义错误。对它们加以区分，可以更快地找到错误。

### 语法错误

语法指的是程序的结构以及此结构的规则。例如，括号必须前后匹配，所以`(1+2)`是合法的，而`8)`就是一个语法错误。

程序中只要出现一处语法错误，Python就会显示出错消息并退出，你的程序就无法运行了。在编程生涯的最初几周中，可能会需要花费

大量时间来查找语法错误。但随着经验的增加，犯错会越来越少，查找起来也会越来越快。

## 运行时错误

第二类错误是运行时错误，这样称呼是因为这种错误只有程序运行后才会出现。这些错误也常被称为**异常**（exception），因为它们常常表示某些异常的（而且不好的）事情发生了。

运行时错误在开头几章中的简单示例里很少会出现，所以可能要过一段时间你才会遇到。

## 语义错误

第三类错误是语义错误，意思是错误与含义相关。如果你的程序中有一个语义错误，程序仍会成功运行，而不会产生任何出错消息，但是它不会执行正确的逻辑。它会做其他的事情。特别需要注意的是，它所做的正是你的代码所告诉它的。

查找语义错误会比较麻烦，因为需要反向查找，查看程序输出并尝试弄明白它到底做了什么。

## 2.9 术语表

变量（variable）：引用一个值的名字。

赋值语句（assignment statement）：将一个值赋值给变量的语句。

状态图（**state diagram**）：用来展示一些变量以及其值的图示。

关键字（**keyword**）：编译器或解释器保留的词，用于解析程序；变量名不能使用关键字，如**if**，**def**，**while**等。

操作数（**operand**）：操作符所操作的值。

表达式（**expression**）：变量、操作符和值的组合，可以表示一个单独的结果值。

求值（**evaluate**）：对表达式按照操作的顺序进行计算，求得其结果值。

语句（**statement**）：表示一个命令或动作的一段代码。至今我们见过赋值语句和打印语句。

执行（**execute**）：运行一条语句，看它说的是什么。

交互模式（**interactive mode**）：使用Python解释器的一种方式，在提示符之后键入代码。

脚本模式（**script mode**）：使用Python解释器的一种方式，从脚本中读入代码并运行它。

脚本（**script**）：保存在文件中的程序。

操作顺序（**order of operations**）：当表达式中有多个操作符和操作对象要求值时，用于指导求值顺序的规则。

拼接（**concatenate**）：将两个操作数首尾相连。

注释（**comment**）：代码中附加的注解信息，用于帮助其他程序员阅读代码，并不影响程序的运行。

语法错误（**syntax error**）：程序中的一种错误，导致它无法进行语法解析（因此也无法被解释器执行）。

异常（**exception**）：程序运行中发现的错误。

语义（**semantics**）：程序表达的含义。

语义错误（**semantic error**）：程序中的一种错误，导致程序所做的事情不是程序员设想的。

## 2.10 练习

### 练习2-1

重申上一张的建议，每当你学习新语言特性时，都应当在交互模式中进行尝试，并故意犯下错误，看会有哪些问题。

- 我们已经见过  $n = 42$  是合法的。那么  $42 = n$  呢？
- 那么  $x = y = 1$  呢？
- 有些语言中，每个语句都需要以分号（`;`）结尾。如果你在Python语句的结尾放一个分号，会有什么情况？
- 如果在语句结尾放的是句号呢？
- 在数学标记中，对于  $x$  乘以  $y$ ，可以这么表达： $xy$ 。在Python中这样尝试会有什么结果？



## 练习2-2

把Python解释器当作计算器来进行练习。

1. 半径为 $r$ 的球体的体积是 $(4/3)\pi r^3$ 。半径为5的球体体积是多少？
2. 假设一本书的定价是24.95美元，但是书店打了40%的折扣（6折）。运费是一本3美元，每加一本加75美分。60本书的总价是多少？
3. 如果我6:52时离开家，并以慢速（6分10秒/千米）跑1.6千米，接下来以4分30秒/千米的速度跑4.8千米，再以慢速跑1.6千米。请问我回家吃早餐是什么时候？

## 第3章 函数

在程序设计中，**函数**是指用于进行某种计算的一系列语句的有名称的组合。定义一个函数时，需要指定函数的名称并写下一系列程序语句。之后，就可以使用名称来“调用”这个函数。

### 3.1 函数调用

前面我们已经见过**函数调用**的一个例子：

```
>>> type(42)
<class 'int'>
```

这个函数的名称是**type**，括号中的表达式我们称之为函数的**参数**。这个函数调用的结果是求得参数的类型。

我们通常说函数“接收”参数，并“返回”结果。这个结果也称为**返回值**（return value）。

Python提供了一些可将某个值从一种类型转换为另一种类型的函数。**int**函数可以把任何可以转换为整型的值转换为整型；如果转换失败，则会报错：

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

`int` 可以将浮点数转换为整数，但不会做四舍五入操作，而是直接舍弃小数部分。

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

`float` 函数将整数和字符串转换为浮点数：

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

最后，`str` 函数将参数转换为字符串：

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

## 3.2 数学函数

Python有一个数学计算模块，提供了大多数常用的数学函数。**模块**（`module`）是指包含一组相关的函数的文件。

要想使用模块中的函数，需要先使用**`import`** 语句将它导入运行环境：

```
>>> import math
```

这个语句将会创建一个名为`math`的**模块对象**（module object）。如果显示这个对象，可以看到它的一些信息：

```
>>> math
<module 'math' (built-in)>
```

模块对象包含了该模块中定义的函数和变量。若要访问其中的一个函数，需要同时指定模块名称和函数名称，用一个句点（.）分隔。这个格式称为**句点表示法**（dot notation）。

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

上面第一个例子使用了`math.log10`来计算以分贝为单位的信号/噪声比（假设`signal_power`和`noise_power`都已经事先定义好了）。`math`模块也提供了`log`函数，用来计算底为`e`的自然对数。

第二个例子计算`radians`的正弦值。这个变量名已经暗示了，`sin`以及`cos`、`tan`等三角函数接受的参数是以弧度（radians）为单位的。若要将角度转换为弧度，可以除以180再乘以 $\pi$ ：

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

表达式`math.pi`从`math`模块中获得变量`pi`。这个变量的值是 $\pi$ 的浮点近似值，大约精确到15位数字。

如果了解三角函数，可以把上面的结果和2的平方根的一半进行比较：

```
>>> math.sqrt(2) / 2.0  
0.707106781187
```

## 3.3 组合

到现在为止，我们已经分别了解了程序的基本元素——变量、表达式和语句，但还没有接触如何将它们有机地组合起来。

程序设计语言最有用的特性之一就是可以将各种小的构建块（building block）**组合**起来。例如，函数的参数可以是任何类型的表达式，包括算术操作符：

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

甚至还包括函数调用：

```
x = math.exp(math.log(x+1))
```

基本上，在任何可以使用值的地方，都可以使用任意表达式，只有一个例外：赋值表达式的左边必须是变量名称，在左边放置任何其他表达式都是语法错误（后面我们还会看到这条规则的例外情况）。

```
>>> minutes = hours * 60          # 正确  
>>> hours * 60 = minutes          # 错误！  
SyntaxError: can't assign to operator
```

---

## 3.4 添加新函数

至此，我们都只是在使用Python提供的函数，其实我们也可以自己添加新的函数。**函数定义** 指定新函数的名称，并提供一系列程序语句，当函数被调用时，这些语句会顺序运行。

下面是一个例子：

```
def print_lyrics():
    print ("I'm a lumberjack, and I'm okay.")
    print ("I sleep all night and I work all day.")
```

**def** 是关键字，表示接下来是一个函数定义。这个函数的名称是 **print\_lyrics**。函数名称的书写规则和变量名称一样：字母、数字和下划线是合法的，但第一个字符不能是数字。关键字不能作为函数名，而且我们应尽量避免函数和变量同名。

函数名后的空括号表示它不接收任何参数。

函数定义的第一行称为**函数头**（header），其他部分称为**函数体**（body）。函数头应该以冒号结束，函数体则应当整体缩进一级。依照惯例，缩进总是使用4个空格，函数体的代码语句行数不限。

本例中**print** 语句里的字符串使用双引号括起来。单引号和双引号的作用相同。大部分情况下，人们都使用单引号，只在本例中这样的特殊情况下才使用双引号。本例中的字符串里本身就存在单引号（这里的单引号作为缩略符号用）。

代码中所有的引号（包括双引号和单引号）都必须是“直引号”，通常在键盘上的**Enter**键附近。而“斜引号”，在Python中是非法的。

如果在交互模式里输入函数定义，则解释器会输出省略号（...）提示用户当前的定义还没有结束：

```
>>> def print_lyrics():  
...     print("I'm a lumberjack, and I'm okay.")  
...     print("I sleep all night and I work all day.")  
... 
```

想要结束这个函数的定义，需要输入一个空行。

定义一个函数会创建一个函数对象，其类型是'**function**'。

```
>>> print(print_lyrics)  
<function print_lyrics at 0xb7e99e9c>  
>>> type(print_lyrics)  
<class 'function'>
```

调用新创建的函数的方式，与调用内置函数是一样的：

```
>>> print_lyrics()  
I'm a lumberjack, and I'm okay.  
I sleep all night and I work all day.
```

定义好一个函数之后，就可以在其他函数中调用它。例如，若想重复上面的歌词，我们可以写一个**repeat\_lyrics**函数：

```
def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()
```

然后可以调用`repeat_lyrics`：

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

当然，这首歌其实并不是这么唱的。

## 3.5 定义和使用

将前面一节的代码片段整合起来，整个程序就像下面这个样子：

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

这个程序包含两个函数定义：`print_lyrics` 和 `repeat_lyrics`。函数定义的执行方式和其他语句一样，不同的是执行后会创建函数对象。函数体里面的语句并不会立即运行，而是等到函数被调用时才执行。函数定义不会产生任何输出。

你可能已经猜到，必须先创建一个函数，才能运行它。换言之，函数定义必须在函数被调用之前先运行。



作为练习，将程序的最后一行移动到首行，于是函数调用会先于函数定义执行。运行程序并查看会有什么样的错误信息。

现在将函数调用那一行放回到末尾，并将函数`print_lyrics`的定义移到函数`repeat_lyrics`定义之后。这时候运行程序会发生什么？

## 3.6 执行流程

为了保证函数的定义先于其首次调用执行，需要知道程序中语句运行的顺序，即**执行流程**。

执行总是从程序的第一行开始。语句按照从上到下的顺序逐一运行。

函数定义并不会改变程序的执行流程，但应注意函数体中的语句并不立即运行，而是等到函数被调用时运行。

函数调用可以看作程序运行流程中的一个迂回路径。遇到函数调用时，并不会直接继续运行下一条语句，而是跳到函数体的第一行，继续运行完函数体的所有语句，再跳回到原来离开的地方。

这样看似简单，但马上你就会发现，函数体中可以调用其他函数。当程序流程运行到一个函数之中时，可能需要运行其他函数中的语句。而后，当运行那个函数中的语句时，又可能再需要调用运行另一个函数的语句！

幸好Python对于它运行到哪里有很好的记录，所以每个函数执行结束后，程序都能跳回到它离开的地方。直到执行到整个程序的结尾，才会结束程序。

总之，在阅读代码时，并不总应该按照代码书写顺序一行行阅读；有时候，按照程序执行的流程来阅读代码，理解的效果可能会更好。

## 3.7 形参和实参 [1]

前面说到的函数有些需要传入参数 [2]。例如，当调用`math.sin`时，需要传入一个数字作为实参。有的函数需要多个实参：`math.pow`需要两个，分别是基数（base）和指数（exponent）。

在函数内部，实参会赋值给称为形参（parameter）的变量。下面的例子是一个函数的定义，接收一个实参：

```
def print_twice(bruce):  
    print(bruce)  
    print(bruce)
```

这个函数在调用时会把实参的值赋到形参`bruce`上，并将其打印两次。

这个函数对任何可以打印的值都可用。

```
>>> print_twice('Spam')  
Spam  
Spam  
>>> print_twice(42)  
42
```

```
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

内置函数的组合规则，在用户自定义函数上也同样可用，所以我们可以对`print_twice` 使用任何表达式作为实参：

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

作为实参的表达式会在函数调用之前先执行。所以在这个例子中，表达式`'Spam '*4` 和`math.cos(math.pi)` 都只执行一次。

也可以使用变量作为实参：

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

作为实参传入到函数的变量的名称（`michael`）和函数定义里形参的名称（`bruce`）没有关系。函数内部只关心形参的值，而不用关心它在调用前叫什么名字；在`print_twice` 函数内部，大家都叫 `bruce` 。

## 3.8 变量和形参是局部的

在函数体内新建一个变量时，这个变量是**局部的**（local），即它只存在于这个函数之内。例如：

```
def cat_twice(part1, part2):  
    cat = part1 + part2  
    print_twice(cat)
```

这个函数接收两个实参，将它们拼接起来，并将结果打印两遍。下面是一个使用这一函数的例子：

```
>>> line1 = 'Bing tiddle '  
>>> line2 = 'tiddle bang.'  
>>> cat_twice(line1, line2)  
Bing tiddle tiddle bang.  
Bing tiddle tiddle bang.
```

当`cat_twice`结束时，变量`cat`会被销毁。这时再尝试打印它的话，会得到一个异常：

```
>>> print(cat)  
NameError: name 'cat' is not defined
```

形参也是局部的。例如，在`print_twice`函数之外，不存在`bruce`这个变量。

## 3.9 栈图

要跟踪哪些变量在哪些地方使用，有时候画一个**栈图**（stack diagram）会很方便。和状态图一样，栈图可以展示每个变量的值，不同的是它会展示每个变量所属的函数。

每个函数使用一个**帧** 包含，帧在栈图中就是一个带着函数名称的盒子，里面有函数的参数和变量。前面的函数示例的栈图如图3-1所示。

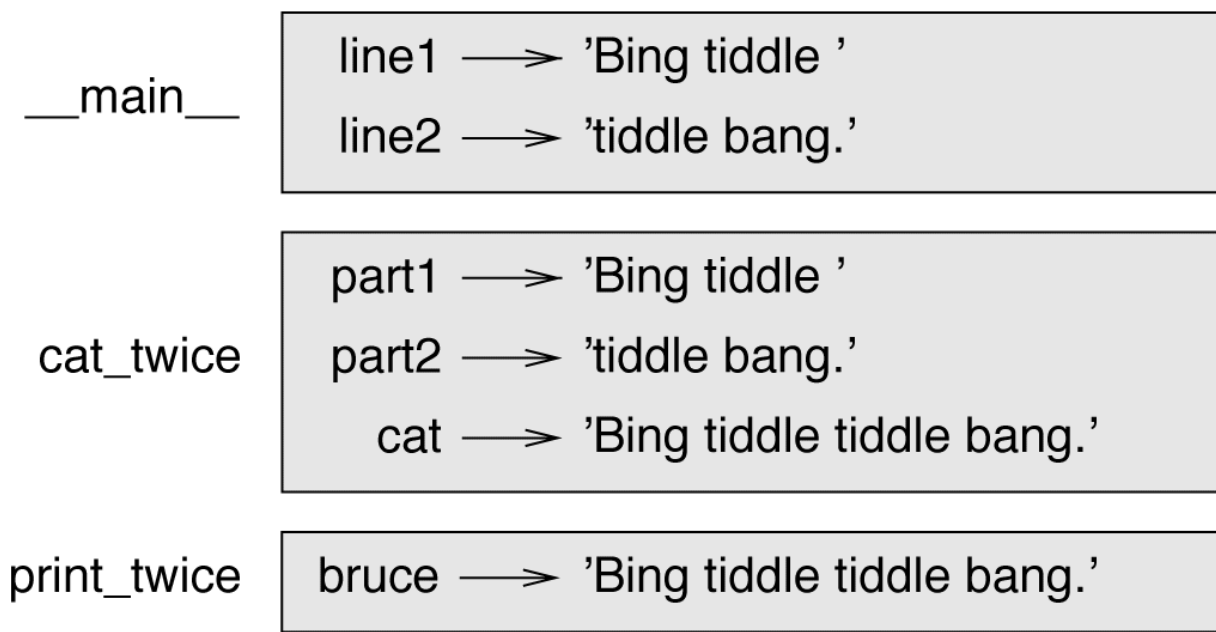


图3-1 栈图

图中各个帧从上到下安排成一个栈，能够展示出哪个函数被哪个函数调用了。在这个例子里，`print_twice` 被 `cat_twice` 调用，而 `cat_twice` 被 `__main__` 调用。`__main__` 是用于表示整个栈图的图框的特别名称。在所有函数之外新建变量时，它就是属于 `__main__` 的。

每个形参都指向与其对应的实参相同的值，所以，`part1` 和 `line1` 的值相同，`part2` 和 `line2` 的值相同，而 `bruce` 和 `cat` 的值相同。

如果调用函数的过程中发生了错误，Python会打印出函数名、调用它的函数的名称，以及调用这个调用者的函数名，依此类推，一直到 `__main__`。

例如，如果在 `print_twice` 中访问 `cat` 变量，则会得到一个 `NameError`：

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print cat
NameError: name 'cat' is not defined
```

上面这个函数列表被称为**回溯**（`traceback`）。它告诉你错误出现在哪个程序文件，哪一行，以及哪些函数正在运行。它也会显示导致错误的那一行代码。

回溯中函数的顺序和栈图中图框的顺序一致。当前正在执行的函数在最底部。

### 3.10 有返回值函数和无返回值函数

在我们使用过的函数中，有一部分函数，如数学函数，会返回结果。因为没有想到更好的名字，我称这类函数为**有返回值函数**

（`fruitful function`）。另一些函数，如 `print_twice`，会执行一个动作，但不返回任何值。我们称这类函数为**无返回值函数**（`void function`）。

当调用一个有返回值的函数时，大部分情况下你都想要对结果做某种操作。例如，你可能会想把它赋值给一个变量，或者用在一个表达式中：

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

在交互模式中调用函数时，Python会直接显示结果：

```
>>> math.sqrt(5)
2.2360679774997898
```

但是在脚本中，如果只是直接调用这类函数，那么它的返回值就会永远丢失掉！

```
math.sqrt(5)
```

这个脚本计算5的平方根，但由于并没有把计算结果存储到某个变量中，或显示出来，所以其实没什么实际作用。

无返回值函数可能在屏幕上显示某些东西，或者有其他的效果，但是它们没有返回值。如果把该结果赋值给某个变量，则会得到一个特殊的值None。

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print result
None
```

值None 和字符串 'None' 并不一样。它是一个特殊的值，有自己独特的类型：

```
>>> print type(None)
<class 'NoneType'>
```

到目前为止，我们自定义的函数都是无返回值函数。再过几章我们就会开始写有返回值的函数了。

## 3.11 为什么要有函数

为什么要花功夫将程序拆分成函数呢？也许刚开始编程的时候这其中的原因并不明晰。下面这些解释都可作为参考。

- 新建一个函数，可以让你有机会给一组语句命名，这样可以让代码更易读和更易调试。
- 函数可以通过减少重复代码使程序更短小。后面如果需要修改代码，也只要修改一个地方即可。
- 将一长段程序拆分成几个函数后，可以对每一个函数单独进行调试，再将它们组装起来成为完整的产品。
- 一个设计良好的函数，可以在很多程序中使用。书写一次，调试一次，复用无穷。

## 3.12 调试

你将会掌握的一个最重要的技能就是调试。虽然调试可能时有烦恼，但它的确是编程活动中最耗脑力、最有挑战、最有趣的部分。



在某种程度上，调试和刑侦工作很像。你会面对一些线索，而且必须推导出事情发生的过程，以及导致现场结果的事件。

调试也像是一种实验科学。一旦猜出错误的可能原因，就可以修改程序，再运行一次。如果猜对了，那么程序的运行结果会符合预测，这样就离正确的程序更近了一步。如果猜错了，则需要重新思考。正如夏洛克·福尔摩斯所说的：“当你排除掉所有的可能性，那么剩下的，不管多么不可能，必定是真相。”（柯南·道尔《四签名》）

对某些人来说，编程和调试是同一件事。也就是说，编程正是不断调试修改直到程序达到设计目的的过程。这种想法的要旨是，应该从一个能做某些事的程序开始，然后做一点点修改，并调试修改，如此迭代，以确保总是有一个可以运行的程序。

例如，Linux是包含了数百万行代码的操作系统，但最开始只是Linus Torvalds编写的用来研究Intel 80386芯片的简单程序。据Larry Greenfield所说：“Linux最早的一个程序是交替打印AAAA和BBBB。后来这些程序演化成了Linux。”（《Linux用户指南》Beta版本1）

### 3.13 术语表

**函数（function）**：一个有名称的语句序列，可以进行某种有用的操作。函数可以接收或者不接收参数，可以返回或不返回结果。

**函数定义（function definition）**：一个用来创建新函数的语句，指定函数的名称、参数以及它包含的语句序列。

函数对象（**function object**）：函数定义所创建的值。函数名可以用作变量来引用一个函数对象。

函数头（**header**）：函数定义的第一行。

函数体（**body**）：函数定义内的语句序列。

形参（**parameter**）：函数内使用的用来引用作为实参传入的值的名称。

函数调用（**function call**）：运行一个函数的语句。它由函数名称和括号中的参数列表组成。

实参（**argument**）：当函数调用时，提供给它的值。这个值会被赋值给对应的形参。

局部变量（**local variable**）：函数内定义的变量。局部变量只能在函数体内使用。

返回值（**return value**）：函数的结果。如果函数被当作表达式调用，返回值就是表达式的值。

有返回值函数（**fruitful function**）：返回一个值的函数。

无返回值函数（**void function**）：总是返回**None**的函数。

**None**：由无返回值函数返回的一个特殊值。

模块（**module**）：一个包含相关函数以及其他定义的集合的文件。

**import**语句 (**import statement**)：读入一个模块文件，并创建一个模块对象的语句。

模块对象 (**module object**)：使用**import** 语句时创建的对象，提供对模块中定义的值的访问。

句点表示法 (**dot notation**)：调用另一个模块中的函数的语法，使用模块名加上一个句点符号，再加上函数名。

组合 (**composition**)：使用一个表达式作为更大的表达式的一部分，或者使用语句作为更大的语句的一部分。

执行流程 (**flow of execution**)：语句运行的顺序。

栈图 (**stack diagram**)：函数栈的图形表达形式，也展示它们的变量，以及这些变量引用的值。

图框 (**frame**)：栈图中的一个图框，表达一个函数调用。它包含了局部变量以及函数的参数。

回溯 (**traceback**)：当异常发生时，打印出正在执行的函数栈。

## 3.14 练习

### 练习3-1

编写一个函数**right\_justify**，接收一个字符串形参**s**，并打印出足够的前导空白，以达到最后一个字符显示在第70列上。

```
>>> right_justify('monty')  
monty
```

提示：可以利用字符串的拼接和重复特性。另外，Python提供了一个内置名为**len**的函数，返回一个字符串的长度，所以**len('allen')**的值是5。

### 练习3-2

函数对象是一个值，可以将它赋值给变量，或者作为实参传递。例如，**do\_twice** 是一个函数，接收一个函数对象作为实参，并调用它两次：

```
def do_twice(f):  
    f()  
    f()
```

下面是一个使用**do\_twice** 来调用一个**print\_spam** 函数两次的示例：

```
def print_spam():  
    print('spam')  
  
do_twice(print_spam)
```

1. 将这个示例存入脚本中并测试它。
2. 修改**do\_twice**，让它接收两个实参，一个是函数对象，另一个是一个值，它会调用函数对象两次，并传入那个值作为实参。

3. 将本章前面介绍的函数`print_twice` 的定义复制到你的脚本中。

4. 使用修改版的`do_twice` 来调用`print_twice` 两次，并传入实参'`spam`'。

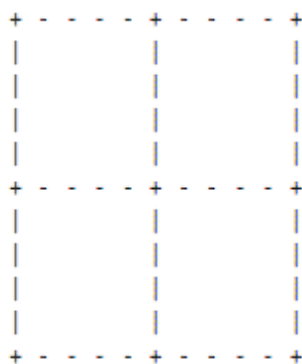
5. 定义一个新的函数`do_four`，接收一个函数对象与一个值，使用这个值作为实参调用函数4次。这个函数的函数体应该只有2条语句，而不是4条。

解答：[http://thinkpython2.com/code/do\\_four.py](http://thinkpython2.com/code/do_four.py)。

### 练习3-3

注意：这个练习应该只用语句和我们已经学过的其他语言特性实现。

1. 编写一个函数，绘制如下的表格：




提示：要在同一行打印多个值，可以使用逗号分隔不同的值：

```
print('+', '-')
```

默认情况下，`Print` 会自动换行，如果你想改变这一行为，在结尾打印一个空格，可以这样做：

```
print('+', end=' ')\nprint('-')
```

这两条语句的输出是 '+ - ' 。

不带参数的`print` 语句会结束当前行并开始下一行 。

2. 编写一个函数绘制类似的表格，但有4行4列 。

解答： <http://thinkpython2.com/code/grid.py> 。鸣谢：这个练习基于 Oualline 的《实践C编程》第3版（O'Reilly Media, 1997）中的一个示例 。

---

[1] 这一段中讲的参数有两种：函数定义里的形参（`parameter`），以及调用函数时传入的实参（`argument`），这里两种是有区分的。——译者注

[2] 调用时传入的参数称为实参（`argument`）。——译者注

## 第4章 案例研究：接口设计

本章通过一个案例研究来展示设计互相配合的函数的过程。

本章介绍**turtle** 模块，通过这个模块可以使用乌龟图形来创造图像。大多数Python安装包都包含了**turtle** 模块，但是如果使用的是PythonAnywhere，则不能直接运行乌龟示例（至少在我写这本书的时候还不行）。

如果你已经在电脑上安装了Python，应该可以运行本章的示例。否则，现在就是安装Python的好时机。我在<http://tinyurl.com/thinkpython2e>上写了安装指南。

本章的代码示例可以从<http://thinkpython2.com/code/polygon.py>下载。

### 4.1 turtle 模块

要检查是否已经安装了**turtle** 模块，可以打开Python解释器并在其中输入：

```
>>> import turtle
>>> bob = turtle.Turtle()
```

运行这段代码时，应该会创建一个新窗口，里面有一个小箭头代表乌龟。请关闭窗口。

创建一个文件`mypolygon.py`，并输入如下代码：

```
import turtle
bob = turtle.Turtle()
print(bob)
turtle.mainloop()
```

`turtle` 模块（以小写`t`开头）提供一个叫作**Turtle** 的函数（以大写`T`开头），它会创建一个**Turtle**对象，我们将其赋值到**bob** 变量。打印**bob** 会得到类似下面的输出：

```
<turtle.Turtle object at 0xb7bfbf4c>
```

这意味着**bob** 变量引用着在**turtle** 模块中定义的**Turtle** 类型的一个**对象**。

`mainloop` 告诉窗口去等待用户进行某些操作，虽然现在除了关闭窗口之外，并没有提供给用户多少有用的操作。

创建好一个乌龟（**Turtle**）之后，就可以调用它的一个**方法**（**method**）来在窗口中移动。方法和函数类似，但是使用的语法略有不同。例如，要让乌龟向前移动：

```
bob.fd(100)
```

这个方法**fd** 和我们称为**bob** 的乌龟对象是关联的。调用方法和发出一个请求类似：你是在请求**bob** 去向前移动。



**fd** 的参数是移动的距离，以像素（**pixel**）为单位，所以实际移动的距离依赖于显示器的分辨率。

**Turtle**对象的其他方法包括**bk**（用于前进和后退）、**lt** 和**rt**（用于左转和右转）。**lt** 和**rt** 的参数是旋转的角度，单位是度。

另外，每只乌龟都拿着一只笔，可以朝上或者朝下；若笔朝下，则会绘制出走过的路迹。方法**pu** 和**pd** 分别表示“笔朝上”（**pen up**）和“笔朝下”（**pen down**）。

若要画一个朝右的角，在程序中（建立**bob** 实例之后，调用**mainloop** 之前）添加如下代码：

```
bob.fd(100)
bob.lt(90)
bob.fd(100)
```

运行这个程序时，将会看到**bob** 先向东走，再向北走，身后留下两条线段。

现在试着修改程序，画出一个正方形来。在成功之前请不要继续！

## 4.2 简单重复

你可能会写下如下代码：

```
bob.fd(100)
bob.lt(90)
```

```
bob.fd(100)
bob.lt(90)

bob.fd(100)
bob.lt(90)

bob.fd(100)
```

使用**for** 语句，可以更紧凑地实现同样功能。把下面的例子加到 `mypolygon.py` 中，并再运行一次：

```
for i in range(4):
    print('Hello!')
```

可能会看到如下输出：

```
Hello!
Hello!
Hello!
Hello!
```

这是**for** 语句的最简单用法，后面我们会看到更多的用法。但这样已经足够重写刚才的画正方形的程序了。请重写后再接着阅读。

下面是使用**for** 语句绘制正方形的程序：

```
for i in range(4):
    bob.fd(100)
    bob.lt(90)
```

**for** 语句的语法和函数定义类似。它也有一个以冒号结束的语句头，并有一个缩进的语句体。语句体可以包含任意数量的语句。

`for` 语句也称为**循环**（`loop`），因为执行流程会遍历语句体，之后从语句体的最开头重新循环执行。在这个例子里，语句体执行了4次。

这个版本的代码和之前的绘制正方形的代码其实还稍有不同，因为在最后一次循环后它多做了一次左转。多余的左转稍微多消耗了点时间，但因为每次循环做的事情都一样，也让代码更简练。这个版本的代码还有一个效果，程序执行完之后，乌龟会回归到初始的位置，并朝向初始相同的方向。

## 4.3 练习

下面是一系列使用乌龟世界的练习。它们力求有趣，但也包含着某些寓意。做这些练习时，可以猜想一下其寓意。

在接下来的章节中有这些练习的解答，所以在完成（或着至少尝试过）之前，请先别继续阅读。

1. 写一个函数**square**，接受一个形参**t**，用来表示一只乌龟。利用乌龟来画一个正方形。

写一个函数调用传入**bob** 作为实参来调用**square** 函数，并再运行一遍程序。

2. 给**square** 函数再添加一个形参**length**。修改函数内容，保证正方形的长度是**length**，并修改函数调用以提供这第二个实参。再运行一遍程序。使用不同的**length** 值测试你的程序。

3. 复制**square**函数，并命名为**polygon**。再添加一个形参**n**并修改函数体以绘制一个正**n**边形。提示：正**n**边形的拐角是 $360/n$ 度。

4. 写一个函数**circle**接受代表乌龟的形参**t**，以及表示半径的形参**r**，并使用合适的长度和边数调用**polygon**画一个近似的圆。使用不同的**r**值来测试你的函数。

**提示：**思考圆的周长（**circumference**），并保证`length * n = circumference`。

另一个提示：如果你觉得**bob**太慢，可以修改**bob.delay**来加速。**bob.delay**代表每次行动之间的停顿，单位是秒。**bob.delay = 0.01**应该能让它跑得足够快。

5. 给**circle**函数写一个更通用的版本，称为**arc**。增加一个形参**angle**，用来表示画的圆弧的大小。这里**angle**的单位是度数，所以当**arc=360**时，则会画一个整圆。

## 4.4 封装

第一个练习要求把画正方形的代码放到一个函数定义中，并将乌龟**bob**作为实参传入，调用该函数。下面是一个解答：

```
def square(t):
    for i in range(4):
        t.fd(100)
        t.lt(90)

square(bob)
```

最内侧的语句，`fd` 和 `lt` 都缩进了两层，表示它们是在 `for` 语句的语句体内部，而 `for` 语句在函数定义的函数体内部。最后一行，`square(bob)`，又重新从左侧开始而没有缩进，这表明 `for` 语句和 `square` 函数的定义都已经结束。

在函数体中，`t` 引用的乌龟和 `bob` 引用的相同，所以 `t.lt(90)` 和直接调用 `bob.lt(90)` 是一样的效果。在这种情况下为什么不直接把形参写为 `bob` 呢？原因是 `t` 可以是任何乌龟，而不仅仅是 `bob`，所以可以再新建一只乌龟，并将它作为参数传入到 `square` 函数：

```
alice = Turtle()  
square(alice)
```

把一段代码用函数包裹起来，称为**封装**（encapsulation）。封装的一个好处是，它给这段代码一个有意义的名称，增加了可读性。另一个好处是，当重复使用这段代码时，调用一次函数比复制粘贴代码要简易得多！

## 4.5 泛化

下一步是给 `square` 函数添加一个 `length` 参数。这里是一个解决方案：

```
def square(t, length):  
    for i in range(4):  
        t.fd(length)  
        t.lt(90)  
  
square(bob, 100)
```

给函数添加参数的过程称为**泛化**（generalization），因为它会让函数变得更通用：在之前的版本中，正方形总是一个大小，而新的版本中，可以是任意大小。

下一步也是一次泛化。我们不再只绘制正方形，而是可以绘制任意边数的多边形。这里是一个方案：

```
def polygon(t, n, length):  
    angle = 360 / n  
    for i in range(n):  
        t.fd(length)  
        t.lt(angle)  
  
polygon(bob, 7, 70)
```

这个例子绘制一个7边形，边长是70。

如果使用的是Python 2，那么angle 的值可能会因为整数除法而错误。一个简单的解决办法是使用`angle = 360.0 / n`。因为分子是一个浮点数，所以结果也会是浮点数。

如果函数的形参比较多，很容易忘掉每一个具体是什么，或者忘掉它们的顺序。所以在Python中，调用函数时可以加上形参名称，这样是合法的，并且有时候会有帮助：

```
polygon(bob, n=7, length=70)
```

这些参数被称为**关键词参数**（keyword argument），因为它们使用“关键词”的形式带上了形参的名称调用（请别和while 与def 之类的Python关键字混淆）。

这个语法使得程序更加可读。它也同样提示了我们实参和形参的工作方式：当调用函数时，实参传入并赋值给形参。

## 4.6 接口设计

下一步是写画圆的`circle`函数，接受形参`r`，表示圆的半径。下面是一个简单的例子，通过调用`polygon`函数画50边的多边形：

```
import math

def circle(t, r):
    circumference = 2 * math.pi * r
    n = 50
    length = circumference / n
    polygon(t, n, length)
```

第一行计算半径为`r`的圆的周长，使用公式 $2\pi r$ 。因为我们使用的是`math.pi`，所以需要先导入`math`模块。依照惯例，`import`语句一般都放在脚本开头。

`n`是我们用于近似画圆的多边形的边数，所以`length`是每个边的长度。因此，`polygon`画出一个50边形，近似于一个半径为`r`的圆。

这个解决方案的缺点之一是`n`是一个常量，因此对于很大的圆，多边形的边线太长，而对于小圆，我们又浪费时间去画过短的边线。解决办法之一是泛化这个函数，加上形参`n`。这样可以给用户（调用`circle`函数的人）更多的控制选择，但接口就不那么清晰整洁了。

函数的**接口**是如何使用它的概要说明：它有哪些参数？这个函数做什么？它的返回值是什么？我们说一个接口“整洁”（`clean`），是说

它能够让调用者完成所想的事情，而不需要处理多余的细节。

在这个例子里，`r` 属于函数的接口，因为它指定了所画的圆的基本属性。相对地，`n` 则不那么适合，因为它说明的是如何画圆的细节信息。

所以与其弄乱接口，不如在代码内部根据周长来选择合适的`n` 值：

```
def circle(t, r):  
    circumference = 2 * math.pi * r  
    n = int(circumference / 3) + 1  
    length = circumference / n  
    polygon(t, n, length)
```

现在多边形的边数是一个接近`circumference/3` 的整数，所以每个边长近似是3，已经小到足够画出好看的圆形，但又足够大到不影响画线效率，并且可接受任何尺寸的圆。

## 4.7 重构

当我写`circle` 函数时，我可以复用`polygon`，因为边数很多的正多边形是圆的很好的近似。但是`arc` 则并不那么容易对付；我们不能使用`polygon` 或者`circle` 来画圆弧。

换个办法，可以先复制一个`polygon` 函数，再通过修改得到`arc` 函数。结果可能类似下面的示例：

```
def arc(t, r, angle):  
    arc_length = 2 * math.pi * r * angle / 360  
    n = int(arc_length / 3) + 1  
    step_length = arc_length / n
```



```
step_angle = angle / n

for i in range(n):
    t.fd(step_length)
    t.lt(step_angle)
```

这个函数的第二部分很像**polygon**的实现，但如果不修改**polygon**的接口，无法直接复用。我们也可以泛化**polygon**函数以接受第三个参数表示圆弧的角度，但那样的话**polygon**（多边形）就不是合适的名称了！所以，我们将这个更泛化的函数称为**polyline**（多边线）：

```
def polyline(t, n, length, angle):
    for i in range(n):
        t.fd(length)
        t.lt(angle)
```

现在我们可以重写**polygon**和**arc**，让它们调用**polyline**：

```
def polygon(t, n, length):
    angle = 360.0 / n
    polyline(t, n, length, angle)

def arc(t, r, angle):
    arc_length = 2 * math.pi * r * angle / 360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

最后，我们可以重写**circle**，改为调用**arc**：

```
def circle(t, r):
    arc(t, r, 360)
```

这个过程——重新组织程序，以改善接口，提高代码复用——被称为**重构**（refactoring）。在这个例子里，我们注意到`arc` 和 `polygon` 中有类似的代码，因此我们把它们的共同之处“重构出来”抽取到`polyline` 函数中。

如果我们早早计划，可能会直接先写下`polyline`，也就避免了重构，但实际上在工程开始时我们往往并没有足够的信息去完美设计所有的接口。开始编码之后，你会更了解面对的问题。有时候，重构正意味着你在编程中掌握了一些新的东西。

## 4.8 一个开发计划

**开发计划**（development plan）是写程序的过程。本章的案例分析中，我们使用的过程是“封装和泛化”。这个过程的具体步骤是：

1. 最开始写一些小程序，而不需要函数定义。
2. 一旦程序成功运行，识别出其中一段完整的部分，将它封装到一个函数中，并加以命名。
3. 泛化这个函数，添加合适的形参。
4. 重复步骤1到步骤3，直到得到一组可行的函数。复制粘贴代码，以避免重复输入（以及重复调试）。
5. 寻找可以使用重构来改善程序的机会。例如，如果发现程序中几处地方有相似的代码，可以考虑将它们抽取出来做一个合适的通用函数。

这个过程也有一些缺点——我们会在后面看到其他方式——但如果在开始编程时不清楚如何将程序分成适合的函数，这样做会带来帮助。这个方法能让你一边开发一边设计。

## 4.9 文档字符串

**文档字符串**（docstring）是在函数开头用来解释其接口的字符串（doc是“文档”documentation的缩写）。下面是一个示例：

```
def polyline(t, n, length, angle):  
    """Draws n line segments with the given length and  
    angle (in degrees) between them. t is a turtle.  
    """  
    for i in range(n):  
        t.fd(length)  
        t.lt(angle)
```

依照惯例，所有的文档字符串都使用三引号括起来。三引号字符串又称为多行字符串，因为三引号允许字符串跨行表示。

文档字符串很简洁，但已经包含了其他人需要知道的关于函数的基本信息。它简明地解释了函数是做什么的（而不涉及如何实现的细节）。它解释了每个形参对函数行为的影响效果以及每个形参应有的类型（如果其类型并不显而易见）。

编写这类文档是接口设计的重要部分。一个设计良好的接口，也应当很简单就能解释清楚；如果你发现解释一个函数很困难，很可能表示该接口有改进的空间。

## 4.10 调试

---

函数的接口，作用就像是函数和调用者之间签订的一个合同。调用者同意提供某些参数，而函数则同意使用这些参数做某种工作。

例如，`polyline` 需要4个参数：`t` 必须是一个Turtle；`n` 必须是整数；`length` 应当是个正数；而`angle` 则必须是一个数字，并且按照度数来理解。

这些需求被称为**前置条件**，因为它们应当在函数开始执行之前就保证为真。相对地，函数结束的时候需要满足的条件称为**后置条件**。后置条件包含了函数预期的效果（如画出线段）以及任何副作用（如移动乌龟或者引起其他改变）。

满足前置条件是调用者的职责。如果调用者违反了一个（文档说明清晰的！）前置条件，因而导致函数没有正确运行，则bug是在调用者，而不在函数本身。

如果前置条件已经满足，但后置条件没有满足，那么bug就出现在函数本身。如果前置和后置前置都定义清晰，可以帮助调试。

## 4.11 术语表

**方法（method）**：与某个对象相关联的一个函数，使用句点表达式调用。

**循环（loop）**：程序中的一个片段，可以重复运行。

**封装（encapsulation）**：将一组语句转换为函数定义的过程。

泛化（**generalization**）：将一些不必要的具体值（如一个数字）替换为合适的通用参数或变量的过程。

关键词参数（**keyword argument**）：调用函数时，附带了参数名称（作为一个“关键词”来使用）的参数。

接口（**interface**）：描述函数如何使用的说明。包括函数的名称，以及形参与返回值的说明。

重构（**refactoring**）：修改代码并改善函数的接口以及代码质量的过程。

开发计划（**development plan**）：写程序的过程。

文档字符串（**docstring**）：在函数定义开始处出现的用于说明函数接口的字符串。

前置条件（**precondition**）：在函数调用开始前应当满足的条件。

后置条件（**postcondition**）：在函数调用结束后应当满足的条件。

## 4.12 练习

### 练习4-1

在<http://thinkpython2.com/code/polygon.py>下载本章的代码。

1. 画一个栈图来显示函数**circle(bob, radius)**运行时的程序状态。你可以手动计算，或者在代码中添加一些**print**语句。

2. 在4.7节中的`arc`函数并不准确，因为使用多边形模拟近似圆，总是会在真实的圆之外。因此，`Turtle`画完线之后会停在偏离正确的目标几个像素的地方。我的解决方案里展示了一种方法可以减少这种错误的效果。阅读代码并考虑是否合理。如果你自己画图，可能会发现它是如何生效的。

### 练习4-2

写一组合适的通用函数，用来画出图4-1所示的花朵图案。

解答：<http://thinkpython2.com/code/flower.py>，另外也需要<http://thinkpython2.com/code/polygon.py>。

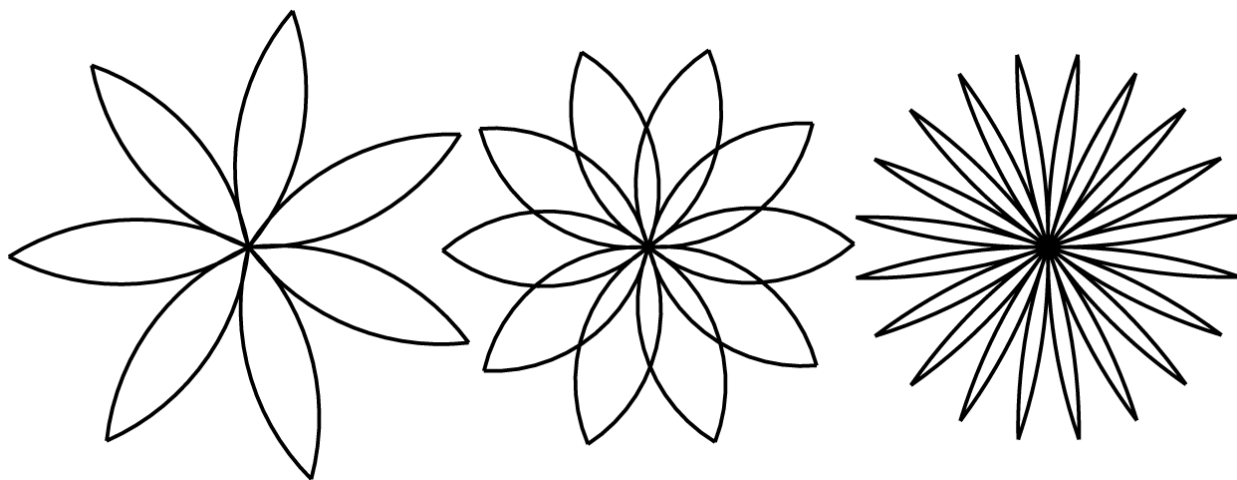


图4-1 花朵图案

### 练习4-3

写一组合适的通用函数，用来画出图4-2所示的图形。

解答：<http://thinkpython2.com/code/pie.py>。

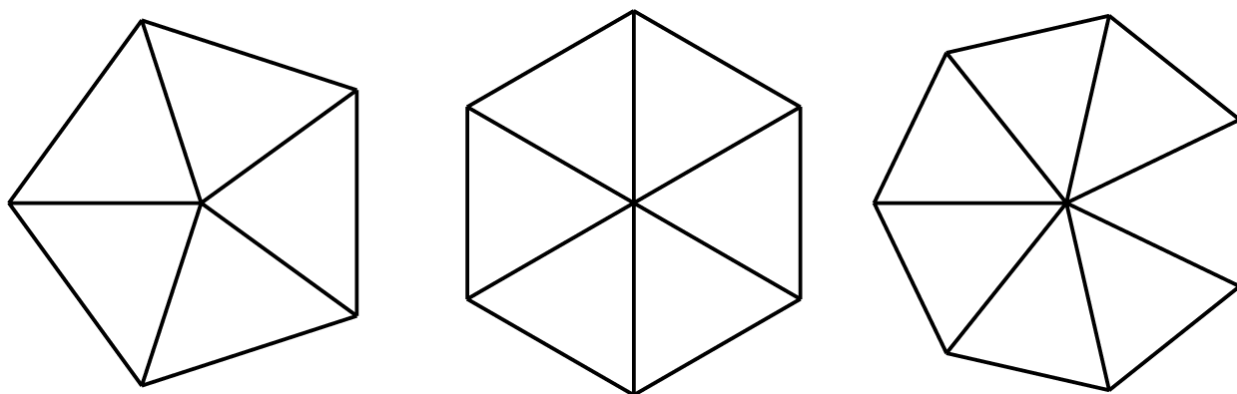


图4-2 饼图

### 练习4-4

字母表中的字母可以使用一些基本元素来构成，如横线、竖线以及一些曲线。设计一个字母表，可以使用最少的基本元素画出来，并编写函数来画出字母。

你应当给每个字母单独写一个函数，名称为`draw_a`、`draw_b`等，并把这些函数放到`letters.py`文件中。可以从<http://thinkpython2.com/code/typewriter.py> 下载一个“乌龟打字机”程序来帮助测试你的代码。

你可以在<http://thinkpython2.com/code/letters.py>获得解答，另外也需要<http://thinkpython2.com/code/polygon.py>。

### 练习4-5

在<http://en.wikipedia.org/wiki/Spiral>阅读关于螺旋线（spiral）的信息；接着编写一段程序来画出阿基米德螺旋（或者其他的某种螺旋线）。

解答: <http://thinkpython2.com/code/spiral.py> °



## 第5章 条件和递归

本章的主要话题是`if` 表达式，它根据程序的状态执行不同的代码。但首先我想要介绍两个新操作符：向下取整除法操作符和求模操作符。

### 5.1 向下取整除法操作符和求模操作符

向下取整除法操作符（`//`）对两个数进行除法运算，并向下取整得到一个整数。例如，假设一个电影的播放时长为105分钟，你可能会想知道按小时算这是多长。传统的除法会得到一个浮点数：

```
>>> minutes = 105
>>> minutes / 60
1.75
```

但是，我们在写小时数时通常并不用小数点。向下取整除法，则丢弃小数部分，得到整数的小时数：

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

要求得余数，可以从分钟数中减去1小时：

```
>>> remainder = minutes - hours * 60
>>> remainder
```

45

另一种办法是使用求模操作符（%）将两个数相除，得到余数：

```
>>> remainder = minutes % 60
>>> remainder
45
```

求模操作符其实有很多实际用途。例如，可以用它来检测一个数是不是另一个的倍数——如果  $x \% y$  是0，则  $x$  可以被  $y$  整除。

另外，也可以用它来获取一个数后一位或后几位数字。例如， $x \% 10$  可以得到  $x$  的个位数（10进制）。类似地， $x \% 100$  可以获得最后两位数。

如果使用的是Python 2，除法机制会有所不同。除法操作符（/）在两个操作数都是整数的情况下，实际进行的是向下取整除法操作，而当两个操作数中有一个是浮点数时，则进行的是浮点数除法。

## 5.2 布尔表达式

**布尔表达式** 是值为真或假的表达式。下面的例子中使用了 == 操作符，来比较两个操作对象是否相等。如果相等，则得 **True**，否则是 **False**：

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` 和 `False` 是类型 `bool` 的两个特殊值；它们不是字符串：

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

`==` 操作符是一个**关系操作符**；其他的**关系操作符**有：

```
x != y      # x不等于y
x > y       # x比y大
x < y       # x比y小
x >= y      # x大于或等于y
x <= y      # x小于或等于y
```

虽然你可能对这些操作已经熟悉，但是Python的符号和数学符号还是有些区别的。最常见的错误是使用单等号（`=`）而不是双等号（`==`）。请记住`=`是一个赋值操作符，而`==`是一个关系操作符。另外，不存在`=<` 或者 `=>` 这样的操作符。

## 5.3 逻辑操作符

**逻辑操作符** 有3个：`and`、`or` 和 `not`。这些操作符的语义（意义）和它们在英语中的意思差不多。例如，`x > 0 and x < 10` 只有当`x` 比0大且 比10小时才为真。

`n%2 == 0 or n%3 ==0`，当其中任意一个条件为真时为真，也就是说，数`n` 可以被2或3整除都可以。

最后，`not` 操作符可以否定一个布尔表达式，所以`not (x > y)` 在`x > y` 为假时为真，即当`x` 小于等于`y` 时真。

严格地说，逻辑操作符的操作对象应该都是布尔表达式，但是 Python 并不那么严格。任何非0的数都被解释为`True` 。

```
>>> 42 and True
True
```

这种灵活性可能会很有用，但有时候也会带来一些小困惑。你可能应该避免使用它（除非你很确切地知道自己在做什么）。

## 5.4 条件执行

为了编写有用的程序，我们几乎总是需要检查条件并据此改变程序的行为的能力。**条件语句** 给了我们这种能力。最简单的形式是`if` 表达式：

```
if x > 0:
    print('x is positive')
```

`if` 之后的布尔表达式被称为**条件**（`condition`）。如果它为真，则之后缩进的语句会运行。否则，什么都不发生。

`if` 表达式的结构和函数定义一样：一个语句头，接着是缩进的语句体。这种类型的语句称为**复合语句**。

语句体中出现的语句数量并没有限制，但是最少需要一行。偶尔可能会遇到需要一个语句体什么都不做（通常是标记一个你还没有来得及写的代码的位置）。这个时候，可以使用`pass` 语句。`pass` 语句什么都不做。

```
if x < 0:
    pass                # TODO: 需要处理负值的情况！
```

## 5.5 选择执行

`if` 语句的第二种形式是**选择执行**，这种形式下，有两种可能，而 `if` 的条件决定哪一种运行。语法看起来是这样的：

```
if x%2 == 0:
    print('x is even')
else:
    print('x is odd')
```

如果`x` 除以2的余数是0，则我们知道`x` 是偶数（**even**），并且程序会显示合适的消息`even'`。如果条件为假，则第二段语句会运行。因为条件必定是真假之一，所以必然只会有一段语句运行。这两段不同的语句称为**分支**（**branch**），因为它们是程序执行流程中的两个支流。

## 5.6 条件链

有时候有超过两种的可能，所以我们需要更多的分支。表达这种计算的一种方式**条件链**（**chained conditional**）：

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

**elif** 是“else if”的缩写。和之前一样，只有一个分支会运行。**elif** 语句的数量没有限制。如果有一个**else** 语句，则它必须放在最后。但也可以没有**else** 语句。

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

每个条件都按顺序检查。如果第一个是**false**，则检查下一个，依此类推。如果有一个条件为真，则运行相应的分支，而整个语句结束。即使有多个条件为真，也只有第一个为真的分支会运行。

## 5.7 嵌套条件

条件判断可以再嵌套条件判断。我们可以修改前一节中的示例，如下：

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

外侧的条件语句包含两个分支。第一个分支包含一行简单的语句。第二个分支则包含了另一个**if** 语句，它本身也有两个分支。这两个分支也都是简单语句，虽然它们其实也可以是条件语句。

虽然语句的缩进让结构非常明晰，但**嵌套条件语句** 会很快随着嵌套层数增多而变得非常难以阅读。应该尽量避免它。

逻辑操作符常常能够用来简化嵌套条件语句。例如，我们可以将下面的语句替换为单独的一个条件：

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

**print** 语句只有在两个条件语句都通过时才运行，所以我们可以使用**and** 操作符达到相同的效果：

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

对于这种类型的条件，**Python**还提供了一个更简洁的语法：

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

## 5.8 递归

函数调用另外一个函数是合法的；函数调用自己也是合法的。这样做有什么好处可能还不明显，但它其实是程序能做的最神奇的事情

之一。例如，考虑下面的函数：

```
def countdown(n):  
    if n <= 0:  
        print('Blastoff!')  
    else:  
        print(n)  
        countdown(n - 1)
```

如果 $n$  是0或负数，它会输出单词“Blastoff!”，其他情况下，它会输出 $n$ ，并调用一个名为`countdown` 的函数——它自己——并传入实参 $n-1$ 。

我们调用这个函数时会发生什么？

```
>>> countdown(3)
```

`countdown` 的执行从 $n=3$  开始，因为 $n$  比0大，所以会输出3，并接着调用自己.....

`countdown` 的执行从 $n=2$  开始，因为 $n$  比0大，所以会输出2，并接着调用自己.....

`countdown` 的执行从 $n=1$  开始，因为 $n$  比0大，所以会输出1，并接着调用自己.....

`countdown` 的执行从 $n=0$  开始，因为 $n$  不比0大，所以会输出单词“Blastoff!”，并返回。

接收 $n=1$  的函数`countdown` 返回。



接收 $n=2$  的函数countdown 返回。

接收 $n=3$  的函数countdown 返回。

然后就会到了\_\_main\_\_ 函数。所以，全部的输出如下：

```
3
2
1
Blastoff!
```

调用自己的函数称为**递归的**（recursive）函数，这个执行的过程叫作**递归**（recursion）。

另外举一个例子，我们可以写一个函数打印某个字符串 $n$  次。

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

如果 $n \leq 0$ ，return 语句会直接退出当前函数。执行流程会立即返回到调用者，之后的语句不会运行。

函数另外的部分和countdown 类似：如果 $n$  大于0，它会打印 $s$  并且调用自己，以再进行 $n-1$ 次显示 $s$  的操作。所以输出的行数是 $1+(n-1)$ ，也就是 $n$ 。

对于这样简单的例子来说，可能使用for 循环会更容易。但我们会在后面见到一些示例，使用for 循环很难写，但使用递归则会很简

单，所以早早开始了解递归是件好事。

## 5.9 递归函数的栈图

在3.10节中，我们使用一个栈图来表示程序在进行函数调用时的状态。同样的栈图，可以用来帮助我们解释递归函数。

一个函数每次被调用时，Python会创建一个帧（function frame），来包含函数的局部变量和参数。对于递归函数，栈上可能同时存在多个函数帧。

图5-1展示了countdown 函数在n=3 调用时的栈图。

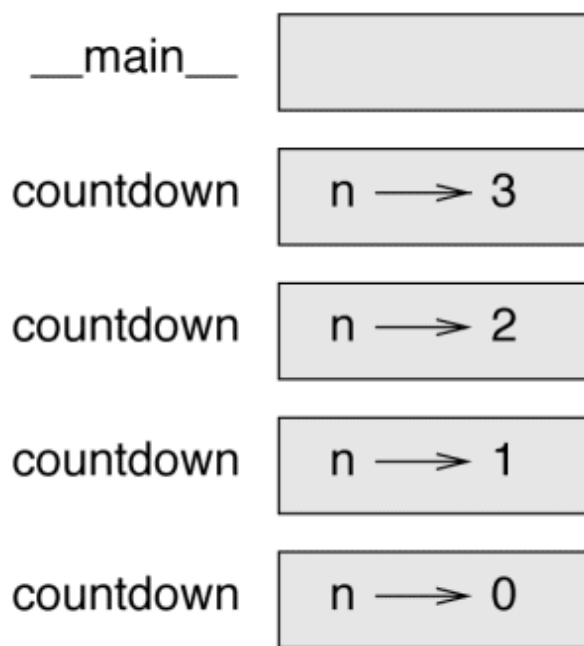


图5-1 栈图

和往常一样，栈的顶端是\_\_main\_\_ 的函数帧。因为我们没有在\_\_main\_\_ 函数里新建任何变量或传入任何参数，所以它是空的。

4个countdown 函数帧有不同的参数n 值。最底端的栈，其n=0，被称为**基准情形**（base case）。因为它不再进行递归调用，所以后面没有其他函数帧了。

作为练习，为函数print\_n 画一个栈图，其调用实参是s = 'Hello' 和n=2。然后写一个函数do\_n，接受一个函数对象和一个数字n 作为形参。它会调用给定的函数n 次。

## 5.10 无限递归

如果一个递归永远达不到基准情形，则它会永远继续递归调用，而程序也永不停止。这个现象被称为**无限递归**，而它并不是个好主意。下面是一个会引起无限递归的最简单函数：

```
def recurse():  
    recurse()
```

在大多数程序环境中，无限递归的函数并不会真的永远执行。Python会在递归深度到达上限时报告一个出错消息：

```
File "<stdin>", line 2, in recurse  
File "<stdin>", line 2, in recurse  
File "<stdin>", line 2, in recurse  
:  
:  
:  
File "<stdin>", line 2, in recurse  
RuntimeError: Maximum recursion depth exceeded
```

这个调用回溯比上一章看到的要大一些。当这个错误发生时，栈上已经有1000个`recurse`帧了！

如果你不小心写出了一个无限循环，请复查自己的函数，确认里面至少有一个基准情形不进行递归调用。如果已经有了一个基准情形，检查是否已经确保在运行时能达到它。

## 5.11 键盘输入

目前为止我们写过的程序都还不能接收用户的输入。它们只能每次做相同的事情。

Python提供了一个内置函数`input`来从键盘获取输入并等待用户输入一些东西。当用户按下回车键，程序会恢复运行，而且`input`则通过字符串形式返回用户输入的内容。在Python 2里，这个函数叫`raw_input`。

```
>>> text = input()
Whate are you waiting for?
>>> text
Whate are you waiting for?
```

在从用户那里获得输入之前，最好打印一个提示信息，告诉用户希望他们输入什么。`raw_input`函数可以接受一个参数作为提示：

```
>>> name = input('What...is your name?\n')
What...is your name?
Authur, King of the Britons!
>>> name
Authur, King of the Britons!
```

提示信息最后的`\n` 表示一个**换行符**，它是会引起输出显示换行的特殊字符。这也是为何用户的输入显示在提示信息的下一行的原因。

如果希望用户输入一个整数，可以尝试将输入值转换为`int`：

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

但如果用户输入不是数字的话，会得到错误：

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

后面我们会看到如何处理这种错误。

## 5.12 调试

当发生语法错误和运行时错误时，出错消息包含了大量的信息，但有时候反而会信息过量。最有用的信息是：

- 错误的类型；
- 发生错误的地方。

语法错误通常都很容易定位，但也有棘手之处。空格问题引起的错误很难处理，因为空格和制表符都是不可见的，我们已经习惯于忽视它们。

```
>>> x = 5
>>> y = 6
      File "<stdin>", line 1
        y = 6
        ^
IndentationError: unexpected indent
```

这个例子中，问题的原因是第二行多缩进了一个空格。但出错消息指向的是`y`，容易误导。总的来说，出错消息会告诉我们发现错误的地址，但真正发生的地方可能在更前面的代码中，有时候甚至在前一行。

运行时错误也是如此。假设你想要按照分贝来计算信噪比。公式是 $SNR_{db} = 10 \lg(P_{\text{signal}} / P_{\text{noise}})$ 。在Python中，你可能会这么写：

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

在运行这个程序时，会得到一个异常：

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

出错消息指向第5行，但那一行其实没有什么错误。要找到真正的错误，可能需要打印出`ratio`的值，结果你会发现是0。问题出在第4行，这里使用了向下取整除法而不是浮点数除法。

你应该花一些时间认真阅读出错消息，但不要认为出错消息上说的每一样都对。

## 5.13 术语表

向下取整除法（**floor division**）：用`//`表示的操作符，用于将两个数相除，并对结果进行向下取整（靠近0取整），得到整数结果。

求模操作符（**modulus operator**）：用`%`表示的操作符，用于两个整数，返回两数相除的余数。

布尔表达式（**boolean expression**）：一种表达式，其值是**True**或**False**。

关系操作符（**relational operator**）：用来表示两个操作对象的比较关系的操作符，如下之一：`==`、`!=`、`>`、`<`、`>=`和`<=`。

逻辑操作符（**logical operator**）：用来组合两个布尔表达式的操作符，有3个：`and`、`or`和`not`。

条件语句（**conditional statement**）：依照某些条件控制程序执行流程的语句。

条件 (**condition**)：条件语句中的布尔表达式，由它决定执行哪一个分支。

复合语句 (**compound statement**)：一个包含语句头和语句体的语句。语句头以冒号 (:) 结尾。语句体相对语句头缩进一层。

分支 (**branch**)：条件语句中的一个可能性分支语句段。

条件链语句 (**chained conditional**)：一种包含多个分支的条件语句。

嵌套条件语句 (**nested conditional**)：在其他条件语句的分支中出现的条件语句。

返回语句 (**return statement**)：导致一个函数立即结束并返回到调用者的语句。

递归 (**recursion**)：在当前函数中调用自己的过程。

基准情形 (**base case**)：递归函数中的一个条件分支，里面不会再继续递归调用。

无限递归 (**infinite recursion**)：没有基准情形的递归，或者永远无法达到基准情形的分支的递归调用。最终，这种无限递归会导致运行时错误。

## 5.14 练习

### 练习5-1



`time` 模块提供了一个函数，名字也叫`time`，它能返回从“纪元”起到当前的格林尼治时间。“纪元”其实是人为选作基准点的时间。在UNIX系统中，纪元时间点是1970年1月1日。

```
>>> import time
>>> time.time()
1437746094.5735958
```

编写一个脚本，读取当前时间，并转换为一天中的小时数、分钟数、秒数，以及从纪元起到现在的天数。

## 练习5-2

费马大定理是说对于任何大于2的 $n$ ，不存在任何正整数 $a$ 、 $b$ 和 $c$ 能够满足：

$$a^n + b^n = c^n$$

1. 编写一个函数`check_fermat`，接收4个形参（即 $a$ 、 $b$ 、 $c$ 和 $n$ ）并检查费马定理是否成立。如果 $n$ 比2大并且满足

$$a^n + b^n = c^n$$

则程序应当打印“天哪，费马弄错了！”，否则程序应当打印“不，那样不行”。

2. 编写一个函数，提示用户输入 $a$ 、 $b$ 、 $c$ 和 $n$ 的值，将它们转换为整数，并使用`check_fermat`来检查它们是否违背了费马定理。

## 练习 5-3

如果给你3根木棍，你可能可以将它们摆成一个三角形，也可能不可以。例如，如果一根木棍的长度是12英寸，而其他两根都只有1英寸，那么你无法让短的木棍在中间相接。对于任意3个长度，有一个简单的测试可以检验它们是否可能组成一个三角形：

如果其中有任何一个长度的值大于其他两个长度的和，则你不能组成三角形。否则可以。（如果两个长度的和等于第三个，则它们组成一个“退化”的三角。）

1. 编写一个函数`is_triangle`，接收3个整数参数，并根据这组长度的木棍是否能组成三角形来打印“Yes”或“No”。

2. 编写一个函数提示用户输入3根木棍的长度，将其转换为整数，并使用`is_triangle`检查这些长度的木棍是否可以组成三角形。

### 练习5-4

下面的程序的输出是什么？画一个栈图来显示程序打印结果的时候的状态。

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)
recurse(3, 0)
```

1. 如果像`recurse(-1, 0)`这样调用这个函数，会发生什么？

2. 编写一段文档字符串，向人解释清楚要使用这个函数需要知道的东西（并且不多写其他内容）。

3. 接下来的练习需要使用第4章描述的turtle 模块。

### 练习5-5

阅读下面的函数，并看看你能否弄清楚它在做什么（参看第4章中的示例）。接着运行它，看你的理解是否正确。

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

### 练习5-6

科赫曲线（Koch curve）是一个分形，它看起来像图 5-2所示。要绘制一个长度为 $x$  的科赫曲线，你只需要做：

1. 绘制长度为 $x/3$ 的科赫曲线。
2. 向左转 $60^\circ$ 。
3. 绘制长度为 $x/3$ 的科赫曲线。
4. 向右转 $120^\circ$ 。

5. 绘制长度为 $x/3$ 的科赫曲线。
6. 向左转 $60^\circ$ 。
7. 绘制长度为 $x/3$ 的科赫曲线。

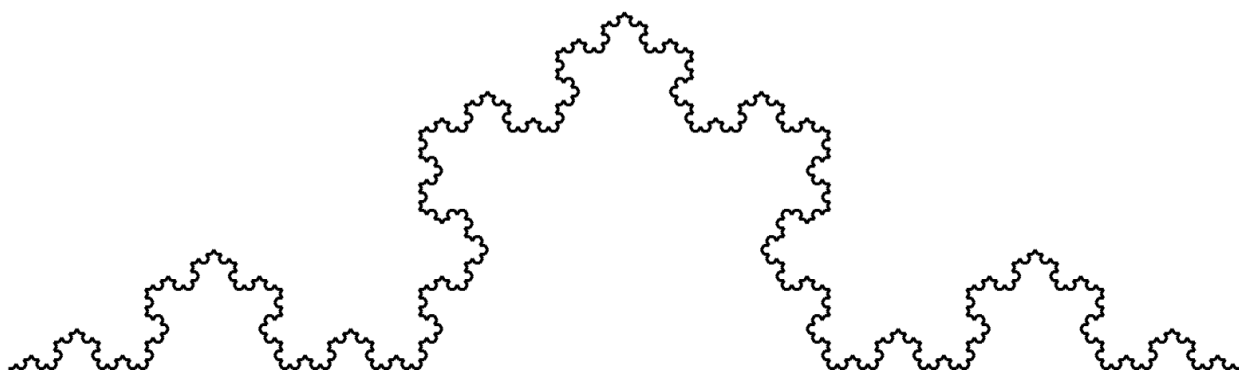


图5-2 一个科赫曲线

当 $x$  比3小的时候例外：在那种情况下，你可以直接绘制一个长度为 $x$  的直线。

1. 编写一个函数**koch**，接收一个**Turtle**以及一个长度作为形参，并使用**Turtle**绘制指定长度的科赫曲线。
2. 编写一个函数**snowflake**，绘制3条科赫曲线，组成一个雪花形状。解答：<http://thinkpython2.com/code/koch.py>。
3. 科赫曲线可以用几种方法泛化。参看[http://en.wikipedia.org/wiki/Koch\\_snowflake](http://en.wikipedia.org/wiki/Koch_snowflake)中的例子，并实现你最喜欢的一个。

## 第6章 有返回值的函数

我们用过的很多Python函数（如数学函数）都会产生返回值。但是，到目前为止我们写的函数都是没有返回值的：它们只产生一个效果，如打印某些值或者移动乌龟，但是并不返回值。在本章中你将学会如何写有返回值的函数。

### 6.1 返回值

调用函数会产生一个返回值，我们一般会将它赋值给一个变量或者用作表达式的组成部分：

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

目前为止我们写的函数都是无返回值的函数。用通俗的话说，它们没有返回值；用更精确的话说，它们返回的值是**None**。

本章中，我们会（终于）写一些有返回值函数。第一个例子是**area**，用于计算给定半径的圆的面积：

```
def area(radius):
    a = math.pi * radius**2
    return a
```

之前我们已经见过`return` 语句，但在有返回值函数中，`return` 语句包含了一个表达式。这个语句的意思是：“立即从这个函数中返回，并使用后面的表达式作为返回值。”表达式可以任意复杂，所以我们可以把这个函数写得更紧凑：

```
def area(radius):  
    return math.pi * radius**2
```

另一方面，类似于`a` 这样的**临时变量** 常常会让调试更容易。

有时候函数中针对不同的条件分支，各有各的返回语句会很有用处：

```
def absolute_value(x):  
    if x < 0:  
        return -x  
    else:  
        return x
```

因为`return` 语句分别在不同的分支中，只有一个运行。

一旦`return` 语句运行，当前的函数就会终结，后面的语句不会执行。`return` 语句之后的代码，或者在其他程序流程永远不可能达到的地方的代码，称为**无效代码**（dead code）。

在有返回函数中，保证每个可能执行路径上都会遇到`return` 语句，是个很好的主意。例如：

```
def absolute_value(x):  
    if x < 0:  
        return -x  
    if x > 0:
```

```
return x
```

这个函数并不正确，因为如果 $x$ 正好是0，则两个条件都不为`true`，则此时函数会没有遇到`return`语句就终结了。如果执行流程到了函数的结尾，返回值是`None`，它并不是0的绝对值。

```
>>> absolute_value(0)
None
```

顺便说一下，Python内置提供了计算绝对值的函数`abs`。

作为练习，写一个`compare`函数，带两个参数 $x$ 和 $y$ ，如果 $x > y$ ，返回1，如果 $x == y$ ，返回0，如果 $x < y$ ，返回-1。

## 6.2 增量开发

当你写更复杂的函数时，可能会发现需要更多的时间来调试。为了对应不断增加的程序复杂度，你可能会想尝试一下称为**增量开发**的过程。增量开发的目标是通过每次只增加和测试一小部分代码，来避免长时间的调试过程。

例如，假设你想要查找两点之间的距离，给定坐标 $(x_1, y_1)$ 和 $(x_2, y_2)$ 。根据毕达哥拉斯定理，距离是：

$$\text{距离} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

第一步考虑Python中`distance`函数应该是什么样子的。换句话说，输入（形参）是什么？输出（返回值）是什么？

在这个例子中，输入是两个点，并可以用4个数字来表示。返回值是距离，它用一个浮点数表示。

现在就可以写出函数的轮廓了：

```
def distance(x1, y1, x2, y2):  
    return 0.0
```

显然，现在这个版本计算的并不是距离；它总是返回0。但它是语法结构正确的，并且能运行，即意味着你可以在继续开发更复杂的功能之前对它进行初步的测试。

要测试这个新函数，使用样本参数调用它：

```
>>> distance(1, 2, 4, 6)  
0.0
```

我选择这些值，因为这样两个点之间，横向距离是3，纵向距离是4；也就是，结果是5（3-4-5直角三角形的斜边）。当测试一个函数时，事先知道正确的结果是很有用的。

到这个时候我们已经确认函数的语法形式是正确的，紧接着可以给函数体添加代码了。合理的下一个步骤是找到距离差 $x_2 - x_1$ 和 $y_2 - y_1$ 。下一版本的函数将这两个距离差保存到临时变量中并打印出来。

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    print('dx is', dx)  
    print('dy is', dy)  
    return 0.0
```



如果函数正确执行，应该会显示‘dx is 3’和‘dy is 4’。如果确实如此，我们就可以确认函数正确地获得了实参，并正确地执行了第一步计算。如果不是如此，则只有几行代码需要检查。

下一步我们计算dx和dy的平方和：

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquared = dx**2 + dy**2  
    print('dsquared is: ', dsquared)  
    return 0.0
```

同样地，你可以在这里再运行一遍程序，并检查输出（应该是25）。最后，可以使用`math.sqrt`来计算并返回结果：

```
def distance(x1, y1, x2, y2):  
    dx = x2 - x1  
    dy = y2 - y1  
    dsquared = dx**2 + dy**2  
    result = math.sqrt(dsquared)  
    return result
```

如果这个函数运行正确，那么你的任务就完成了。否则，你可能需要在`return`语句之前打印出`result`的值。

最终版本的函数运行时并不打印任何东西；它只会返回一个值。我们之前写的`print`语句在调试时很有用，但一旦你的函数编写正确，就应该删除掉它们。这种代码称为**脚手架代码**（scaffolding），因为它们在构建程序的过程中很有用，但并不是最终产品的一部分。

开始的时候，应当每次只添加一到两行代码。当你获得更多经验之后，就会发现自己可以编写和调试更多的代码了。不论如何，增量开发都能帮你节省大量的调试时间。

这个过程有以下几个关键点。

1. 以一个可以正确运行的程序开始，每次只做小的增量修改。如果在任意时刻发现错误，你都应当知道错在哪里。
2. 使用临时变量保存计算的中间结果，你可以显示和检查它们。
3. 一旦整个程序完成，你可能会想要删除掉某些脚手架代码或者把多个语句综合到一个复杂表达式中。但只在不会增加代码阅读的难度时才应该那么做。

作为练习，使用增量开发来编写一个函数`hypotenuse`，给定直角三角形的另外两个直角边的长度时，它返回斜边的长度。开发过程中，记录每一步的情况。

## 6.3 组合

你可能已经想到，在一个函数中可以调用另外一个函数。作为示例，我们会写一个函数，它接收两个点，圆心和圆周上的一点，并计算圆的面积。

假设圆心保存在变量`xc`和`yc`中，而圆周上的点保存在`xp`和`yp`上。第一步是算出圆的半径，也就是这两个点的距离。我们刚才已经写了一个函数，`distance`，正好有这个功能：

```
radius = distance(xc, yc, xp, yp)
```

第二步是使用上一步算出来的半径来计算圆的面积。我们刚才也写了这个函数：

```
result = area(radius)
```

将这两步封装成一个函数，我们得到：

```
def circle_area(xc, yc, xp, yp):  
    radius = distance(xc, yc, xp, yp)  
    result = area(radius)  
    return result
```

临时变量`radius`和`result`在开发和调试时有用，可一旦程序已经可以正确运行，我们就可以使用函数组合来简化函数：

```
def circle_area(xc, yc, xp, yp):  
    return area(distance(xc, yc, xp, yp))
```

## 6.4 布尔函数

函数可以返回布尔值，这样可以很方便地隐藏函数内复杂的检测。例如：

```
def is_divisible(x, y):  
    if x % y == 0:  
        return True  
    else:  
        return False
```

通常布尔函数的命名都类似于是/否的问句。`is_divisible` 返回 `True` 或 `False`，表示 `x` 是否可以被 `y` 整除。

这里是一个例子：

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

`==` 操作符的结果是一个布尔值，所以我们可以把这个函数写得更加紧凑：

```
def is_divisible(x, y):
    return x % y == 0
```

布尔函数常常用在条件语句中：

```
if is_divisible(x, y):
    print('x is divisible by y')
```

你可能想这么写：

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

但这里多出来的比较是不必要的。

作为练习，写一个函数 `is_between(x, y, z)`，当  $x \leq y \leq z$  时，返回 `True`，其他情况返回 `False`。

## 6.5 再谈递归

至今为止，我们只涉及Python的一个很小的子集，但你可能会有兴趣知道，这个子集已经是一个完备的编程语言，也就是说，任何可以计算的问题，都可以用这个子集语言来完成。任何已有的程序，都可以用现在已经学会的语言特性重写出来（实际上，可能还需要一些命令来控制诸如键盘、鼠标、光盘之类的设备，但仅此而已）。

要证明这个论断，并不是简单的工作。这个证明最早是由第一代计算机科学家之一阿兰·图灵（Alan Turing）完成的（有人会争辩他其实是个数学家，但大部分早期的计算机科学家都是从数学家开始的）。因此，这被称为图灵论题（Turing Thesis）。若想了解关于图灵论题的更完整（更准确）的讨论，我推荐Michael Sipser的《计算理论导引》（*Introduction to the Theory of Computation* , Course Technology, 2012）一书。

为了初步了解如何使用我们现在学会的工具，可以考虑几个递归定义的数学函数。递归定义和循环定义有些相似，因为同样地，定义中都会包含要定义的事物本身。真正的循环定义往往没什么用：

**vorpai:**

一个形容词，用来描述一个vorpai的事物。

如果你在词典中看到这样的定义，可能会感觉恼怒。另一方面，如果你查看阶乘函数（用!表示）的定义，可能会看到如下内容：

$$0! = 1$$

$$n! = n(n-1)!$$

这个定义说明0的阶乘是1，而任意其他值 $n$ 的阶乘是 $n-1$ 的阶乘乘以 $n$ 。

所以3!是3乘以2!，而2!是2乘以1!，而1!是1乘以0!。综合起来，3!等于3乘以2乘以1乘以1，即6。

如果能够使用递归定义来描述一个事物，那么也可以使用Python程序来计算它。第一步是决定使用什么形参。在这个例子里，很明显函数**factorial**需要一个整数形参：

```
def factorial(n):
```

如果实参正好是0，我们只需要直接返回1：

```
def factorial(n):  
    if n == 0:  
        return 1
```

否则，接下来是有意思的地方，我们需要递归调用函数来计算 $n-1$ 的阶乘，并乘以 $n$ ：

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        recurse = factorial(n-1)  
        result = n * recurse  
        return result
```

这个程序的运行流程和5.8节里的**countdown** 函数类似。如果我们使用实参值3调用**factorial**：

因为3不是0，我们使用第二个分支，计算**n-1** 的阶乘.....

因为2不是0，我们使用第二个分支，计算**n-1** 的阶乘.....

因为1不是0，我们使用第二个分支，计算**n-1** 的阶乘.....

因为0等于0，我们使用第一个分支并返回1，不再需要进行任何递归调用了。

返回值（1）乘以**n=1**，结果返回。

返回值（1）乘以**n=2**，结果返回。

返回值（2）乘以**n=3**，结果是6，而这个结果就是整个函数的返回值。

图6-1显示了这一系列函数调用的栈图。

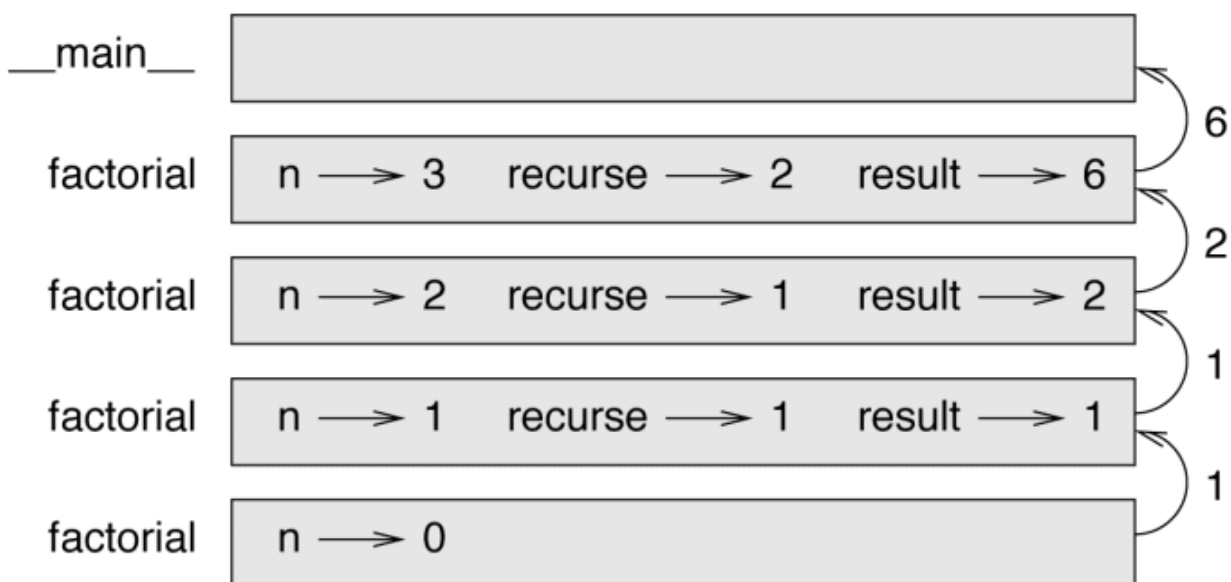


图6-1 栈图

结果值在图中显示为沿着栈向上回传。在每个帧中，返回值是 `result` 的值，即 `n` 和 `recurse` 的乘积。

最后一帧中，局部变量 `recurse` 和 `result` 不存在，因为新建它们的分支并没有运行。

## 6.6 坚持信念

跟踪程序执行的流程是阅读程序的一个办法，但那样很快就会陷入迷宫境况。另外有个办法，我称为“坚持信念”。在遇到一个函数调用时，不去跟踪执行的流程，而假定函数是正确工作的，能够返回正确的结果。

事实上，在使用内置函数时，你已经在这样尝试着坚持信念了。当调用 `math.cos` 或 `math.exp` 时，你并不去检查那些函数的内部实



现。你只会假定它们是正确的，因为写这些内置函数的一定是很优秀的程序员。

当调用自己写的函数时，这个道理也成立。例如，在6.4节中，我们写了`is_divisible`函数用来判断一个数是否可以被另一个数整除。一旦我们说服自己认定这个函数是正确的——通过检查代码和测试——就可以直接使用它，而不需要再细看内部实现了。

对递归函数来说，也是如此。当调用递归函数时，不需要检查执行的流程，你应该假定递归调用是正确的（返回正确的结果），并问自己：“假设我能够正确得到 $n-1$ 的阶乘，如何计算 $n$ 的阶乘？”很明显你可以做到，直接乘以 $n$ 即可。

当然，在还没有完成函数的编写时，就假设它能正确工作，看起来有些奇怪，但那也正是为什么我称它为“坚持信念”的原因！

## 6.7 另一个示例

除阶乘`factorial`之外，最常见的递归数学定义是斐波那契数列（`fibonacci`），其定义如下（参见[http://en.wikipedia.org/wiki/Fibonacci\\_number](http://en.wikipedia.org/wiki/Fibonacci_number)）：

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

翻译成Python后，看起来是这样：

```
def fibonacci (n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

如果你在这里试图跟踪执行的流程，即使是很小的参数 $n$ ，都会感觉头都要炸了。但因为坚持信念，如果你假定两个递归调用都正常工作，那么很明显，把它们加到一起必然得到正确的结果。

## 6.8 检查类型

如果我们调用**factorial** 函数，并给定1.5作为实参，会发生什么呢？

```
>>> factorial(1.5)  
RuntimeError: Maximum recursion depth exceeded
```

看起来像是无限递归。怎么会这样？函数中有一个基准情形——当 $n == 0$ 时。但如果 $n$ 不是整数，我们就可能错过这个基准情形，而永远递归下去。

在第一个递归调用中， $n$ 是0.5。第二个， $n$ 是-0.5。从此以后，它会越来越小（更小的负数），但永远不会变成0。

我们有两个选择。可以尝试泛化函数**factorial**，使之能正确处理浮点数，或者我们也可以让**factorial**检查其实参的类型。第一个选择在数学上叫作伽玛函数（**gamma function**），它有些超出了本书的范围。所以我们选择第二个。

我们可以使用内置函数**isinstance**来检查实参的类型。与此同时，我们也可以确保实参是正数：

```
def factorial (n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

第一个基准情形处理非整数，第二个处理负数。这两种情况中，程序打印一个错误信息，并返回**None**，表示出现了问题：

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

如果我们通过了这两个测试，就能确保知道**n**是正数或0，所以我们可以证明递归必然终结。

这个程序演示了一个模式，它有时被称为**守卫**（guardian）。前两个条件就像守卫一样，保护后面的代码，以免出现错误。守卫使得证明代码的正确性成为可能。

在11.3节中我们会看到一个更灵活的方案，用以打印错误信息：抛出一个异常。

## 6.9 调试

将一个大程序分解为小函数，自然而然地引入了调试的检查点。如果一个函数不能正常工作，可以考虑3种可能性。

- 函数获得的实参有问题，某个前置条件没有达到。
- 函数本身有问题，某个后置条件没有达到。
- 函数的返回值有问题，或者使用的方式不正确。

要排除第一种可能，可以在函数开始的地方加上`print` 语句，显示实参的值（以及它们的类型）。或者可以添加代码来显式地检查前置条件。

如果实参看起来没错，在每个`return` 语句前添加`print` 语句，显示返回值。如果有可能，手动检查返回值。考虑使用能更容易检验结果的实参来调用函数，就像6.2节中的那样。

如果函数看起来正常，检查调用它的代码，确保返回值被正确使用（或者确实被使用了！）。

在函数的开端和结尾处增加**print** 语句，能帮助我们更清晰地了解函数的执行流程。例如，这里是一个添加了**print** 语句的 **factorial** 函数：

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

**space** 是一个字符串，包含多个空格，用来控制输出内容的缩进。下面是调用**factorial(4)** 的结果：

```
                factorial 4
            factorial 3
        factorial 2
    factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
```

如果你对函数调用的流程有困惑，这种输出可以帮你。开发有效的脚手架代码需要花费时间，但一点点脚手架可以节省大量的调试。

## 6.10 术语表

临时变量（**temporary variable**）：在复杂计算中用于保存中间计算值的变量。

无效代码（**dead code**）：程序中的一些代码，永远不可能运行。常常是写在**return** 语句之后的代码。

增量开发（**incremental development**）：一个程序开发计划，通过每次只增加少量代码，并加以测试的步骤，来减少调试。

脚手架代码（**scaffolding**）：在开发过程中使用的，但在最终版本中不需要的代码。

守卫（**guardian**）：一个编程模式。使用条件语句来检查并处理可能产生错误的情形。

## 6.11 练习

### 练习6-1

为下面的程序绘制一个栈图。程序的输出是什么？

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square
```

```
x = 1
y = x + 1
print c(x, y+3, x+y)
```

## 练习 6-2

Ackermann函数,  $A(m, n)$ , 定义如下:

$$A(m, n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

参考[http://en.wikipedia.org/wiki/Ackermann\\_function](http://en.wikipedia.org/wiki/Ackermann_function)。编写一个函数`ack`, 计算Ackermann函数的值。使用你的函数求`ack(3, 4)`的值, 它应当是125。对于很大的数字 $m$ 和 $n$ , 会发生什么?

解答: <http://thinkpython2.com/code/ackermann.py>。

## 练习 6-3

回文是一个正向和逆向拼写都相同的单词, 例如“noon”和“redivider”。递归地说, 如果一个单词第一个和最后一个字母相同, 并且中间是一个回文, 则该单词是回文。

下面的函数接收一个字符串形参并返回第一个、最后一个以及中间的字母:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]
```

```
def middle(word):  
    return word[1:-1]
```

在第8章中查看它们是如何使用的。

1. 将这些函数保存到一个文件 `palindrome.py` 中并测试它们。如果你使用一个包含两个字母的字符串调用 `middle`，会发生什么？使用一个字母呢？空字符串呢？空字符串写作 `''` 并且不包含任意字母。

2. 编写一个函数 `is_palindrome`，接收一个字符串形参，并当它是回文的时候返回 `True`，否则返回 `False`。记着你可以使用内置函数 `len` 来检测字符串的长度。

解答：[http://thinkpython2.com/code/palindrome\\_soln.py](http://thinkpython2.com/code/palindrome_soln.py)。

## 练习 6-4

我们说一个数  $a$  是  $b$  的乘方，如果  $a$  可以被  $b$  整除，并且  $a/b$  也是  $b$  的乘方。编写一个函数 `is_power` 接收形参  $a$  和  $b$ ，当  $a$  是  $b$  的乘方时返回 `True`。注意：你需要考虑基准情形。

## 练习 6-5

$a$  和  $b$  的最大公约数（GCD）是它们两个都能整除的最大的数。

寻找两个数的最大公约数的方法之一是基于如下观察：如果  $r$  是  $a$  除以  $b$  的余数，则  $\text{gcd}(a, b) = \text{gcd}(b, r)$ 。作为基准情形，我们可以使用  $\text{gcd}(a, 0) = a$ 。



编写一个函数gcd，接收形参a和b，并返回它们的最大公约数。

鸣谢：这个练习是基于Abelson和Sussman的《计算机程序的构造和解释》（*Structure and Interpretation of Computer Programs*）（MIT出版社，1996）一书。

## 第7章 迭代

本章讲关于迭代的话题。迭代即重复运行一段代码语句块的能力。我们在5.8节见过一种使用递归来进行的迭代，在4.2节中见过另一种使用**for** 循环进行的迭代。在本章中我们将会看到使用**while** 循环进行的第三种迭代。首先我们先进一步讲讲变量赋值的话题。

### 7.1 重新赋值

你应当已经发现，对一个变量进行多次赋值是合法的。新的赋值语句使现有的变量引用一个新值（并不再引用老值）。

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

因为第一次显示**x** 时，它的值是5，而第二次时它的值是7。

图7-1显示了在状态图中，**重新赋值** 是什么样子的。

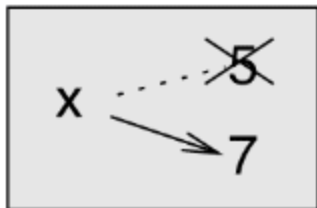


图7-1 状态图

在这里我想解释一个常见的误区。因为Python使用等号(=)来表示赋值，故很容易将`a = b`这样的赋值语句错误理解为数学中表示`a`和`b`相等的命题。这样理解是错误的。

首先，相等判断是个对称的关系，而赋值并不是。例如，在数学中，如果`a=7`那么`7=a`。但是在Python中，语句`a = 7`是合法的，但`7 = a`则是非法的。

另外，在数学中，一个相等判断的命题总是非真即假。如果现在`a=b`，那么`a`总会等于`b`。在Python中，赋值语句会让两个变量变得相等，但它们并不会总保持那个状态：

```
>>> a = 5
>>> b = a      # a和b现在相等了
>>> a = 3      # a和b不再相等了
>>> b
5
```

第三行修改`a`的值，但是并不会修改`b`的值，所以它们不再相等。

虽然重新赋值常常很有用处，但是应该谨慎使用。如果变量的值经常变化，会导致程序难以阅读和调试。

## 7.2 更新变量

重新赋值的最常见形式是**更新**，此时变量的新值依赖于旧值。

```
>>> x = x + 1
```

---

这个语句的意思是“获取x 的当前值，加一，再更新x 为此新值”。

如果尝试更新一个并不存在的变量，则会得到错误，因为Python 在赋值给x 之前会先计算等号右边的部分：

```
>>> x = x + 1
NameError: name 'x' is not defined
```

在更新变量之前，必须先对它进行**初始化**。通常通过一个简单赋值操作来进行初始化：

```
>>> x = 0
>>> x = x + 1
```

通过加1来更新一个变量，称为**增量**（increment）；减1的操作称为**减量**（decrement）。

## 7.3 while 语句

计算机常被用来自动化重复处理某些任务。重复执行相同或相似的任务，而不犯错误，这是电脑所擅长于人之处。在计算机程序中，重复也被称为**迭代**。

我们之前已经看到两个函数countdown 和print\_n，它们使用递归来进行迭代操作。由于迭代如此常见，Python提供了语言特性来

支持它。其中一个是在4.2节中见过的**for** 循环语句。后面我们会再回到这个话题。

另一个则是**while** 语句。下面是使用**while** 语句实现的**countdown** 函数：

```
def countdown(n):  
    while n > 0:  
        print(n)  
        n = n - 1  
    print('Blastoff!')
```

你基本上可以按照英语来理解**while** 语句。它的意思是：“每当**n** 还大于0时，显示**n** 的值，并将**n** 减1。当**n** 变成0的时候，显示单词 **Blastoff!**。”

用更正式的说法，下面是**while** 语句执行的流程。

1. 确定条件是真还是假。
2. 如果条件为假，退出**while** 语句，并继续执行后面的语句。
3. 如果条件为真，则运行**while** 语句的语句体，并返回第1步。

这种类型的流程称为**循环**（loop），因为第3步又循环返回到最顶端的第1步了。

循环的语句体里面应当修改一个或多个变量的值，以致循环的条件最终能变成假，而退出循环。否则这个循环会永远重复下去，这样

的情况叫作**无限循环**。计算机科学家在读到洗发液的说明“涂抹、冲洗、重复”时，总会感到有趣，因为这是一个无限循环。

在**countdown** 这个例子里，我们可以证明循环必然终结：如果 **n** 是0或负数，该循环从不运行。否则，**n** 的值都会减小，因此最终**n** 会变成0。

对于某些循环，就并不一定那么容易判断了。例如：

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:           # n是偶数
            n = n / 2
        else:                   # n是奇数
            n = n*3 + 1
```

这个循环的条件是 **n != 1**，所以只要**n** 还没有变成1而导致条件变假，循环就会一直进行下去。

每一个循环中，程序输出**n** 的值，并检查它是偶数还是奇数。如果是偶数，**n** 会除以2。如果是奇数，**n** 会被替换为 **$n*3+1$** 。例如，如果传入**sequence** 函数的参数是3，则**n** 的结果值是：3, 10, 5, 16, 8, 4, 2, 1。

因为**n** 有时候增加，有时候减少，没有办法找到明显的证据确定**n** 一定会最终变成1，或者说程序会终止。对于某些特定的**n** 值，我们可以证明最终会终止。例如，如果开始的参数值是2的幂方，则每次循环

$n$  都是偶数，直到变成1。前面的例子中有一部分就是这样的序列，以16开始。

但困难的问题是，我们是否能够证明这个程序对所有的正值  $n$  都可以最终终止。至今为止，还没有人对这个问题给出证明或证伪！（参见[http://en.wikipedia.org/wiki/Collatz\\_conjecture](http://en.wikipedia.org/wiki/Collatz_conjecture)）。

作为练习，重写5.8节中的`print_n`函数，使用循环而非递归来实现。

## 7.4 break 语句

有时候只有在循环语句体的执行途中才能知道是不是到了退出循环的时机。这时候可以使用**break** 语句来跳出循环。

例如，假设你想要获得用户输入，直到他们输入**done**。可以这么写：

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

循环的条件是**True**，总是为真，所以循环会一直运行，直到遇到**break** 语句。

每次循环之内，都会先用一个尖括号（>）来提示用户输入。如果用户输入**done**，**break** 语句会退出循环。否则程序会显示出用户输入的内容，并重新回到循环的顶端。这里是一个运行的实例：

```
> not done
not done
> done
Done!
```

这种写**while** 循环的方式很常见，因为可以把判断循环条件的逻辑放在循环中的任何地方（而不只是在顶端），并且可以以肯定的语气来表示终结条件（“当这样发生时停止循环”），而不是否定的语气（“继续执行，直到那个条件发生”）。

## 7.5 平方根

程序中常常使用循环来进行数值计算，以一个近似值开始，并迭代地优化计算结果。

例如，计算平方根的方法之一是牛顿方法。假设你想要知道**a** 的平方根。如果你以任意一个估计值**x** 开始，可以使用如下的方程获得一个更好的估计值。

$$y = \frac{x + a/x}{2}$$

例如，如果**a** 是4而**x** 是3:

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
```



```
>>> y
2.16666666667
```

这个结果更接近正确的答案 ( $\sqrt{4} = 2$ )。如果我们使用新的估计值重复这个过程，会得到更近似的结果：

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

再经过几次重复更新，估计值会几乎完全准确：

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

通常来说，我们并不能提前知道需要多少步才能得到正确的答案，但是当估计值不再变化时，我们就知道达到目的了。

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

当 `y == x` 时，可以终止。下面是一个以估计值 `x` 开始，并不断迭代优化直到它不再变化的循环：

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

对于大多数`a` 值来说，这样效果很好，但通常来说，测试`float` 的相等是危险的。浮点数值只是近似正确：大部分有理数，如 $1/3$ ，以及无理数，如 $\sqrt{2}$ ，都不能用`float` 精确表示。

比起判断`x` 和`y` 是否精确相等，更安全的方式是利用内置函数`abs` 来计算它们之间差值的绝对值，或者说量级：

```
if abs(y-x) < epsilon:
    break
```

这里`epsilon` 的值是`0.0000001`，用来决定近似度是足够的。

## 7.6 算法

牛顿方法是**算法** 的一个例子：它是解决一类问题的机械化过程（在这个例子里，问题是计算平方根）。

要理解算法是什么，从一个算不上算法的东西开始可能更简单。在学习个位数相乘时，你可能背诵过乘法表。实际上，你记住了100个特别的答案。这种知识不算是算法。

但是，如果你比较“懒”，可能已经学会了一些小技巧来偷懒。例如，要计算 $n$  和9的乘积，你可以写下 $n-1$ 作为十位数， $10-n$  作为个位

数。这个小技巧是计算任意个位数和9的乘积的通用方案。这算是一个算法！

相似地，你学过的进位加法、借位减法以及长除法都是算法。算法的特点之一是它们不需要任何聪明才智就能执行。它们是一个机械化的过程，其中每一步都依照一组简单的规则接着上一步进行。

执行算法非常枯燥，但设计算法的过程却充满趣味和智力挑战，并且是计算机科学的一个核心部分。

一些人们自然而然、毫无困难或者下意识所做的事情，用算法表达却最为困难。理解自然语言是一个好例子。我们都能理解自然语言，但是至今为止还没有人能解释我们是怎么做到的，至少没办法用算法解释。

## 7.7 调试

当你开始编写更大的程序时，常常会发现自己花费更多的时间用于调试。更多的代码意味着更多的出错机会，以及更多可能隐藏着bug的地方。

削减调试时间的方法之一是“二分调试”（debugging by bisection）。例如，如果你的程序有100行代码，如果每次检查一行，需要100步。

相反地，可以尝试把问题分成两半。找到程序的中点，或者接近那里的地方，找一个可以检验的中间结果。添加一个**print** 语句（或

者其他的可以有检查效果的代码) 并运行程序。

如果中点检验的结果是错误的, 说明错误必然出现在程序的前半部分。如果是正确的, 那错误则在程序的后半部分。

每进行一次这样的检查, 就减少了一半需要检查的代码。经过6步之后(显然少于100步), 就能够减少到一至两行代码, 至少理论上如此。

实践中, 常常很难确定“程序的中点”在哪里, 并且并不总是能够检验它。通过数代码行数来确定中点显然没有意义。相反地, 应当思考程序中哪些地方可能出错, 哪些地方容易加上一个检查。然后选择一个你认为在其前后发生错误概率差不多的点进行检查。

## 7.8 术语表

重新赋值 (reassignment): 对一个已经存在的变量赋予一个新值。

更新 (update): 一种赋值操作, 新值依赖于变量的旧值。

初始化 (initialization): 一种赋值操作, 给变量一个初始的值, 以后可以进行更新。

增量 (increment): 一种更新操作, 增加变量的值(常常是加1)。

减量 (decrement): 一种更新操作, 减少变量的值。

迭代（iteration）：使用递归函数调用或者循环来重复执行一组语句。

无限循环（infinite loop）：一个终止条件永远无法满足的循环。

算法（algorithm）：解决一类问题的通用过程。

## 7.9 练习

### 练习7-1

复制7.5节的循环并封装到一个名为`square_root`的函数中，这个函数接收一个形参`a`。选择一个合理的值`x`，并返回`a`的平方根的估计值。

要测试这个方法，可以编写一个名为`test_square_root`的函数，打印下面这样的表格：

a	mysqrt(a)	math.sqrt(a)	diff
-	-----	-----	----
1.0	1.0	1.0	0.0
2.0	1.41421356237	1.41421356237	2.22044604925e-16
3.0	1.73205080757	1.73205080757	0.0
4.0	2.0	2.0	0.0
5.0	2.2360679775	2.2360679775	0.0
6.0	2.44948974278	2.44948974278	0.0
7.0	2.64575131106	2.64575131106	0.0
8.0	2.82842712475	2.82842712475	4.4408920985e-16
9.0	3.0	3.0	0.0

第一列是一个数，`a`；第二列是数`a`的平方根，使用`mysqrt`函数计算；第三列是使用`math.sqrt`计算出的平方根；第四列是两种计

算结果的差值的绝对值。

## 练习7-2

内置函数`eval` 接收一个字符串并使用Python解释器对它进行求值。例如：

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

编写一个函数`eval_loop`，迭代地提示用户，接收他们的输入并使用`eval` 求值，并打印出结果。

它应当一直继续，直到用户输入 '`done`'，并返回最后一个求值的表达式的结果。

## 练习7-3

数学家拉马努金（Srinivasa Ramanujan）找到了一个无限序列，可以用来生成 $\pi$ 的数值近似值：

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

编写一个函数`estimate_pi`，使用这个公式计算并返回 $\pi$ 的近似估计。它应当使用一个`while` 循环来计算求和的每一项，直到最后一

项的值小于**1e-15**（这是Python对 $10^{-15}$ 的标记法）。你可以通过和`math.pi` 比较来检查计算的结果。

解答：<http://thinkpython2.com/code/pi.py>。

## 第8章 字符串

字符串和整数、浮点数以及布尔类型都不同。字符串是一个**序列**（sequence），即它是一个由其他值组成的有序集合。本章中你将见到如何访问构成字符串的各个字符，并学到字符串类提供的一些方法。

### 8.1 字符串是一个序列

字符串是一个字符的**序列**（sequence）。可以使用方括号操作符来访问字符串中单独的字符：

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

第二个语句选择`fruit` 中的第1个字符，并将它赋值给`letter` 变量。

方括号中的表达式称为**下标**（index）。下标表示想要序列中的哪一个字符（所以用**index**这个名称）。

但你可能发现得到的和预料不一样：

```
>>> letter
'a'
```



对大多数人来说，'banana' 的第一个字母是**b**，而不是**a**。但对计算机科学家来说，下标表示的是离字符串开头的偏移量，而第一个字母的偏移量是0。

```
>>> letter = fruit[0]
>>> letter
'b'
```

所以**b** 是'banana' 的第0个字母，**a** 是第1个，**n** 是第2个。

可以使用包括变量和操作符的表达式作为下标。

```
>>> i = 1
>>> fruit[i]
'a'
>>> fruit[i+1]
'n'
```

但下标的值必须是整数，否则你会得到：

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

## 8.2 len

**len** 是一个内置函数，返回字符串中字符的个数：

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

要获得字符串的最后一个字母，你可能会想这么写：

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

`IndexError` 出现的原因是 `'banana'` 中没有下标为6的字母。因为我们是从0开始计算的，6个字母的下标是0到5。要获得最后一个字符，需要从`length` 里减1：

```
>>> last = fruit[length-1]
>>> last
'a'
```

或者，你可以使用负数下标。负数下标从字符串结尾处倒着数。表达式`fruit[-1]` 返回最后一个字母，表达式`fruit[-2]` 返回倒数第二个字母，依此类推。

## 8.3 使用for 循环进行遍历

有很多计算都涉及对字符串每次处理一个字符的操作。它们常常从开头起，每次选择一个字符，对它做一些处理，再继续，直到结束。这种处理的模式，我们称为**遍历**（traversal）。编写遍历逻辑的方法之一是使用`while` 循环：

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print(letter)
    index = index + 1
```

这个循环遍历字符串，并将每个字符显示在单独的一行上。循环的结束条件是`index < len(fruit)`，所以当`index`等于字符串的长度时，条件为假，循环体不被运行。最后访问的字符下标为`len(fruit)-1`，正好是字符串最后一个字符。

作为练习，写一个函数，接收一个字符串作为形参，并倒序显示它的字母，每个字母单独一行。

写遍历逻辑的另一个方式是使用`for` 循环：

```
for letter in fruit:  
    print(letter)
```

每次迭代之中，字符串中的下一个字符会被赋值给变量`letter`。循环会继续直到没有剩余的字符为止。

下面的示例展示了如何利用字符串拼接（字符串加法）和一个`for` 循环来生成字母序列（也就是，按字母顺序排序的序列）。在Robert McCloskey的书《为小鸭让路》（*Make Way for Ducklings*）中，小鸭们的名字是Jack、Kack、Lack、Mack、Nack、Ouack、Pack及Quack。下面的循环按顺序输出这些名字：

```
prefixes = 'JKLMNOPQ'  
suffix = 'ack'  
  
for letter in prefixes:  
    print(letter + suffix)
```

输出是：

```
Jack  
Kack  
Lack  
Mack  
Nack  
Oack  
Pack  
Qack
```

当然那并不完全正确，因为“Ouack”和“Quack”拼写错了。作为练习，修改程序解决这个问题。

## 8.4 字符串切片

字符串中的一段称为一个**切片**（slice）。选择一个切片和选择一个字符类似：

```
>>> s = 'Monty Python'  
>>> s[0:5]  
'Monty'  
>>> s[6:12]  
'Python'
```

操作符`[n:m]`返回字符串从第`n`个字符到第`m`个字符的部分，包含第`n`个字符，但不包含第`m`个字符。这个行为有些违反直觉，但如果想象下标是指向字符之间的位置，可以帮助我们理解它，如图8-1所示。

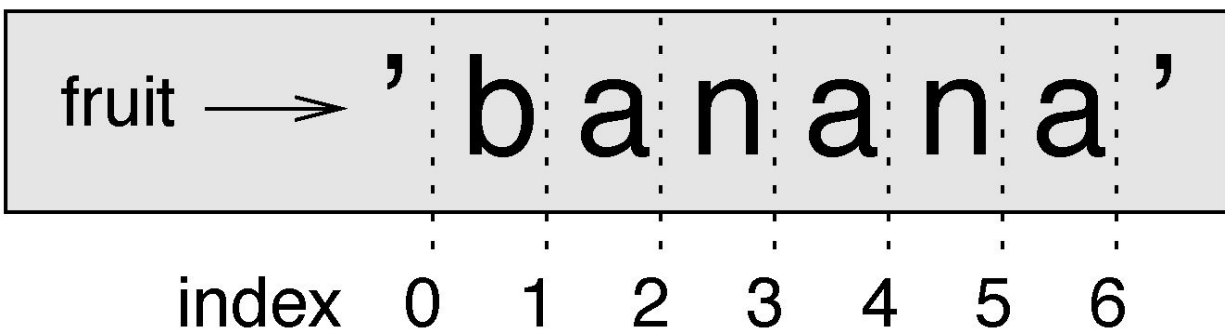


图8-1 切片的下标

如果省略掉第一个下标（冒号之前的那个），切片会从字符串开头开始。如果省略掉第二个下标，切片会继续到字符串的结尾。

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

如果第一个下标大于或等于第二个下标，结果是**空字符串**，用两个引号表示：

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

空字符串不包含任何字符，长度为0，但除此之外，它和其他字符串一样。

继续本例，你认为`fruit[:]`表示什么？尝试一下看看结果。

## 8.5 字符串是不可变的

想要修改字符串的某个字符，你可能会想直接在赋值左侧使用[]操作符。例如：

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

这个例子中的“对象”（object）是字符串，而“项”（item）是指你想要赋值的那个字符。就现在来说，一个**对象**和值是差不多的东西，但我们后面会细谈它（参见10.10节）。

这个错误产生的原因是因为字符串是**不可变**（immutable）的，也就是说，不能修改一个已经存在的字符串。你能做的最多是新建一个字符串，它和原来的字符串稍有不同：

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

这个例子使用新的首字符和greeting的一个切片拼接起来。它对原来的字符串没有影响。

## 8.6 搜索

下面的这段函数是做什么的？

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
```

```
    index = index + 1
    return - 1
```

从某种意义上说，`find` 是`[]` 操作符的反面。和`[]` 操作符通过一个下标查找对应的字符不同，它根据一个字符查找其出现在字符串中的下标。如果没有找到字符，函数返回 **-1** 。

这是我们第一次在循环内部看到`return` 语句。如果 `word[index] == letter`，函数直接跳出循环并立即返回。

如果字符没有出现在字符串中，程序正常退出循环，并返回 **-1** 。

这种计算的模式——遍历一个序列，并当找到我们寻找的目标时返回——称为**搜索**。

作为练习，修改`find` 函数，让它接收第3个形参，表示从`word` 的哪个下标开始搜索。

## 8.7 循环和计数

下面的代码计算字母**a** 在字符串中出现的次数：

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

这个程序展示了另一种计算模式，称为**计数器**。变量`count`初始化为0，接着每次找到一个**a**时计数器加1。当循环结束时，`count`保存着结果——**a**出现的总次数。

作为练习，将这段代码封装成函数**count**，并泛化它以接收字符串和要计数的字母作为形参。

接着重写**count**函数，不直接遍历字符串，而是使用前面一节中的3形参版本的**find**函数。

## 8.8 字符串方法

字符串提供了很多完成各种操作的有用的方法。方法和函数很相似——它接收形参并返回值——但语法有所不同。例如，方法**upper**接收一个字符串，并返回一个全部字母都是大写的字符串。

和函数的语法**upper(word)**不同，它使用方法的调用语法**word.upper()**。

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

这种句点表示法指定了方法的名称，以及方法应用到的字符串的名称**word**。空的括号表示这个方法没有任何参数。



方法的调用称为**invocation** [\[1\]](#)；在这个例子里，我们说我们在**word** 字符串上调用方法**upper** 。

实际上，字符串本来就有一个方法**find**，和我们之前写的**find** 函数非常相似：

```
>>> word = 'banana'
>>> index = word.find('a')
>>> index
1
```

在这个例子中，我们在**word** 上调用**find** 方法，并传入要查找的字母作为实参。

实际上，**find** 方法比我们的函数更通用；它可以用来查找子字符串，而不仅仅是字符：

```
>>> word.find('na')
2
```

默认情况下，**find** 在字符串的开始启动，但它还可以接收第二个实参，表示从哪一个下标开始查找：

```
>>> word.find('na', 3)
4
```

这是可选参数的一个示例。**find** 还可以接收第三个实参，表示查找到哪个下标就结束：

```
>>> name = 'bob'
>>> name.find('b', 1, 2)
```

这个搜索失败，因为**b**并没有在字符串的下标1到2之间（不包括2）出现。**find**在搜索时只搜索到第二个（但不包括第二个）下标为止，这使**find**和切片操作符的行为一致。

## 8.9 操作符in

**in**是一个布尔操作符，操作于两个字符串上，如果第一个是第二个的子串，则返回**True**，否则返回**False**：

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

例如，下面的函数打印出**word1**中出现且出现在**word2**中的所有字母：

```
def in_both(word1, word2):
    for letter in word1:
        if letter in word2:
            print(letter)
```

精心选择变量名称后，Python有时会读起来很像英语。可以这样读这个循环：“for (each) letter in (the first) word, if (the) letter (appears) in (the second) word, print (the) letter”。

下面是用这个函数比较单词apples和oranges的结果：

```
>>> in_both('apples', 'oranges')
a
e
s
```

## 8.10 字符串比较

关系操作符也可以用在字符串上。检查两个字符串是否相等：

```
if word == 'banana':
    print('All right, bananas.')
```

其他的关系操作符在将单词按照字母顺序比较时有用：

```
if word < 'banana':
    print('Your word,' + word + ', comes before banana.')
elif word > 'banana':
    print('Your word,' + word + ', comes after banana.')
else:
    print('All right, bananas.')
```

Python处理大小写字母时和人处理时不一样。所有的大写字母都在小写字母之前。所以：

```
Your word, Pineapple, comes before banana.
```

处理这个问题的常用办法是先将字符串都转换为标准的形式，如都转换成全小写字母形式，再进行比较。如果你遇到一个武装着Pineapple的敌人需要保护自己时，请记住这个办法。

## 8.11 调试

当使用下标来遍历序列中的值时，要正确实现遍历的开端和结尾并不容易。下面是一个函数，能够比较两个单词，如果它们互为倒序，则返回**True**，但这个函数包含了两个错误：

```
def is_reverse(word1, word2):
    if len(word1) != len(word2):
        return False

    i = 0
    j = len(word2)

    while j > 0:
        if word1[i] != word2[j]:
            return False
        i = i+1
        j = j-1

    return True
```

第一个**if** 语句检查两个单词是否长度相同。如果不同，我们就立即返回**False**，否则在后面整个函数中，都可以认为两个单词是相同长度的。这是6.8节中讲到的守卫模式的一个实例。

**i** 和 **j** 是下标：**i** 用于正向遍历**word1**，而**j** 用于反向遍历**word2**。如果我们找到两个不匹配的字母，则可以立即返回**False**。如果完成整个循环后所有的字母仍然都相等，则返回**True**。

如果使用单词“pots”和“stop”来测试这个函数，我们会预期返回值是**True**，但实际上会得到一个**IndexError**：

```
>>> is_reverse('pots', 'stop')
...
File "reverse.py", line 15, in is_reverse
    if word1[i] != word2[j]:
IndexError: string index out of range
```

为了调试这类错误，第一步可以在发生错误的那行代码之前打印出索引的值。

```
while j > 0:
    print i, j      # 在这里打印

    if word1[i] != word2[j]:
        return False
    i = i+1
    j = j-1
```

这样再一次运行程序时，能获得更多的信息：

```
>>> is_reverse('pots', 'stop')
0 4
...
IndexError: string index out of range
```

第一次迭代时，`j` 的值是4，超出了'`pots`'的范围。最后一个字符的下标是3，所以`j`的初始值应该是`len(word2)-1`。

如果修改这个错误并重新运行程序，会得到：

```
>>> is_reverse('pots', 'stop')
0 3
1 2
2 1
True
```

这回我们得到了正确的结果，但看起来循环只运行了3次，有些可疑。为了对具体发生了什么有更清晰的印象，可以画一个状态图。第一个迭代中，`is_reverse`的帧显示在图8-2中。

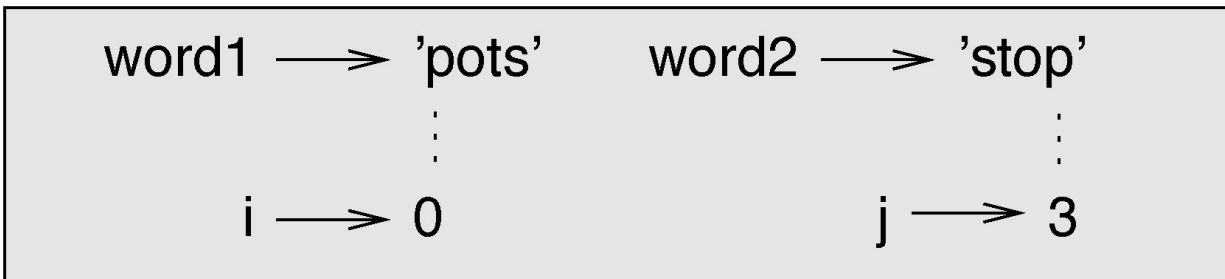


图8-2 状态图

我特意安排了帧中变量的位置，并使用虚线来显示*i* 和*j* 指向 `word1` 和 `word2` 中的字符。

从这个图开始，在纸上运行程序，每个迭代修改*i* 和*j* 的值。找到并修复这个函数的第二个错误。

## 8.12 术语表

对象（`object`）：变量可以引用的一种事物。就现在来说，可以把“对象”当作“值”来使用。

序列（`sequence`）：一个有序的值的集合，其中每个使用一个下标来定位。

项（`item`）：序列中的一个值。

下标（`index`）：用于在序列中选择元素的整数值。例如，可以用于在字符串中选取字符。在Python中下标从0开始。

切片（`slice`）：字符串的一部分，通过一个下标范围来定位。

空字符串（**empty string**）：没有字符，长度为0的字符串，使用一对引号来表示。

不可变（**immutable**）：序列的一种属性，表示它的元素是不可以改变的。

遍历（**traverse**）：迭代访问序列中的每一个元素，并对每个元素进行相似的操作。

搜索（**search**）：一种遍历的模式，当找到它想要的元素时停止。

计数器（**counter**）：一种用来计数的变量，通常初始化为0，后来会递增。

方法调用（**invocation**）：调用一个方法的语句。

可选参数（**optional argument**）：函数或方法中，并不必须有的参数。

## 8.13 练习

### 练习8-1

在<http://docs.python.org/3/library/stdtypes.html#string-methods>阅读字符串方法的文档。你可能会想实验一下其中的一些方法，以确保自己理解了它们的工作方式。**strip** 和 **replace** 特别有用。

文档中使用了一种可能会引起困惑的语法。例如，`find(sub[, start[, end]])` 中的方括号表示可选的参数。所以`sub`是必需的，但是`start`是可选的，并且如果使用了`start`，则`end`是可选的。

## 练习8-2

有一个字符串方法叫作`count`，和我们之前在8.9节中展示的方法类似。阅读这个方法的文档，并写一个程序调用它来计算'`banana`'中`a`出现的次数。

## 练习8-3

字符串切片可以接受第三个下标用来指定“步长”，即相邻的字符之间的距离。步长为2，意思是切片每次取接下来第2个字符；步长3意思是每次取接下来第3个字符，等等。

```
>>> fruit = 'banana'
>>> fruit[0:5:2]
'bnn'
```

步长为-1表示切片按照相反的方向访问字符串，所以切片`[::-1]`会得到一个逆序的字符串。

使用这个特性来编写一个一行版本的`is_palindrome`函数（见练习6-3）。

## 练习8-4



下面的几个函数目的 都是检查一个字符串是否包含小写字母，但至少有一个是错误的。对每个函数，描述一下这个函数到底做了什么（假设形参是一个字符串）。

```
def any_lowercase1(s):
    for c in s:
        if c.islower():
            return True
        else:
            return False

def any_lowercase2(s):
    for c in s:
        if 'c'.islower():
            return 'True'
        else:
            return 'False'

def any_lowercase3(s):
    for c in s:
        flag = c.islower()
    return flag

def any_lowercase4(s):
    flag = False
    for c in s:
        flag = flag or c.islower()
    return flag

def any_lowercase5(s):
    for c in s:
        if not c.islower():
            return False
    return True
```

## 练习8-5

凯撒密码（Caesar Cypher）是一个比较弱的加密形式，它涉及将单词中的每个字母“轮转”固定数量的位置。轮转一个字母意思是在字母表中移动它，如果需要，再从开头开始。所以‘A’轮转3个位置是‘D’，而‘Z’轮转一个位置是‘A’。

要对一个单词进行轮转操作，对其中每一个字母进行轮转即可。例如，“cheer”轮转7位的结果是“jolly”，而“melon”轮转-10位结果是“cubed”。在电影《2001太空漫游》中，舰载机器人叫作HAL，这个单词正是IBM轮转-1位的结果。

编写一个函数`rotate_word`，接收一个字符串以及一个整数作为参数，并返回一个新字符串，其中的字母按照给定的整数值“轮转”位置。

你可以使用内置函数`ord`，它能够将一个字符转换为数值编码，以及函数`chr`，它将数值编码转换为字符。字母表中的字母是按照字母顺序编码的，所以，例如：

```
>>> ord('c') - ord('a')
2
```

因为'c'在字母表中的下标是2。但是请注意：大写字母的数字编码是不同的。

因特网上有些可能冒犯人的笑话是用ROT13编码的。ROT13是轮转13位的凯撒密码。如果你不容易被冒犯，可以寻找一些并解码。

解答：<http://thinkpython2.com/code/rotate.py>。

---

[1] 普通函数的调用，称为`call`。——译者注

## 第9章 案例分析：文字游戏

本章介绍第二个案例分析，讲述的是通过搜索具有某种特性的单词来解决单词谜题这一话题。例如，我们会寻找英语单词中最长的回文单词，还会搜索那些其字母按照字母表顺序排列的单词。另外，我会介绍另一种程序开发计划：缩减问题规模，回归成之前解决过的问题。

### 9.1 读取单词列表

为本章的练习，我们需要准备一个英文单词列表。互联网上有很多可用的单词列表，但最适合我们的目标的单词列表，是由Grady Ward收集整理并作为Moby词典项目（参看[http://wikipedia.org/wiki/Moby\\_Project](http://wikipedia.org/wiki/Moby_Project)）的一部分贡献给公共域的。它包含113 809个正式的填字游戏用词，即那些认为可以用于纵横填字游戏和其他类型文字游戏的单词。在Moby集合中，文件名是113809of.fic；可以从<http://thinkpython.com/code/words.txt>下载一个副本，但文件名是更简单的words.txt。

这个文件是纯文本，所以可以使用文本编辑器打开，也可以使用Python读入它。内置函数open接收文件名作为参数，并返回一个文件对象（file object），可以用来读取文件。

```
>>> fin = open('words.txt')
```

**fin** 是用来表示文件对象作为输入源时常用的名称。文件对象提供了几个方法用于读取内容，包括**readline**，它会从文件里读入字符，直到获得换行符为止，并将读入的结果作为一个字符串返回：

```
>>> fin.readline()
'aa\r\n'
```

在这个特定的列表中，第一个单词是"aa"，它是一种火山熔岩。序列**\r\n**表示两个空格字符，一个是回车，一个是换行，用于把这个单词和其他单词分隔开。

文件对象会记录它读到文件的哪个位置，因此如果再次调用**readline**，会得到下一个单词：

```
>>> fin.readline()
'aah\r\n'
```

下一个单词是"aah"，也是一个完全合法的单词，所以别用奇怪的眼光看着我。或者，如果是那几个空白字符在干扰你，可以使用字符串的方法**strip** 去掉它们：

```
>>> line = fin.readline()
>>> word = line.strip()
>>> word
'aahed'
```

你也可以在**for** 循环中使用文件对象。下面的代码读入**words.txt** 并每行打印出一个单词：

```
fin = open('words.txt')
for line in fin:
    word = line.strip()
    print(word)
```

## 9.2 练习

在下一节里有这些练习的解答。在继续阅读解答之前，应当至少尝试一下每一个练习。

### 练习9-1

编写一个程序，读入`words.txt` 并且打印出那些长度超过20个字符的单词（不算空白字符）。

### 练习9-2

1939年，Ernest Vincent Wright出版了一本5万字的小说*Gadsby*，这本书里没有包含字母“e”。因为“e”是英语中最常见的字母，所以这并不是件容易的事。

实际上，不使用这最常见的字母的话，仅仅是构建一条单独的构思也是很难的事情。开始时会很慢很艰难，但保持谨慎和长时间的训练，你可以渐渐掌握方法。

好吧，我先停下来。 [\[1\]](#)

写一个函数`has_no_e`，当给定的单词不包含字母“e”时，返回`True`。

修改前一节练习中的代码，打印出不含“e”的单词，并计算这种单词在整个单词表中的百分比。

### 练习9-3

编写一个函数**avoids**，接收一个单词，以及一个包含禁止字母的字符串，当单词不含任何禁止字母时，返回**True**。

修改你的程序，提示用户输入包含禁止字母的字符串，并打印出不包含任意禁止字母的单词的个数。能不能找到一组5个禁止字母的组合，它们排除的单词最少？

### 练习9-4

编写一个名为**uses\_only**的函数，接收一个单词以及字母组成的字符串，当单词只由这些字母组成时返回**True**。你可以造一个句子，其单词只由字母**acefhlo**组成吗？除了“**Hoe alfalfa**”之外呢？

### 练习9-5

编写一个名为**uses\_all**的函数，接收一个单词以及由需要的字母组成的字符串，当单词中所有需要的字母都出现了至少一次时返回**True**。有多少单词使用了所有的元音字母**aeiou**？而**aeiouy**呢？

### 练习9-6

编写一个名叫**is\_abecedarian**的函数，如果单词中的字母是按照字母表顺序排列的（两个重复字母也可以），则返回**True**。有

多少这样的单词？

## 9.3 搜索

前面一节的所有练习都有一个共同点；它们可以使用我们在8.6节中介绍的搜索模式来解决。最简单的例子是：

```
def has_no_e(word):
    for letter in word:
        if letter == 'e':
            return False
    return True
```

`for` 循环遍历单词`word` 中的字符。如果我们找到字母“e”，可以立即返回`False`；否则只能继续下一个字母。如果正常退出了循环，则说明我们没有找到“e”，所以返回`True`。

使用`in` 操作符，可以把这个函数写得更简洁。上面这个示例没有写得更简洁是因为想要展现搜索模式的逻辑。

`avoids` 是`has_no_e` 的更通用的版本，它们的结构相同：

```
def avoids(word, forbidden):
    for letter in word:
        if letter in forbidden:
            return False
    return True
```

一旦发现一个禁止的字母，可以立即返回`False`；如果运行到循环结束，则返回`True`。

`uses_only` 函数也类似，只是它条件判断的意思是相反的：

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

它接收的参数并不是一个禁止字母列表，而是一个可用字母列表 `available` 。如果我们发现单词中遇到了并不属于 `available` 的字母，则可以返回 `False` 。

`uses_all` 函数也类似，但单词和字母列表的角色相反。

```
def uses_all(word, required):
    for letter in required:
        if letter not in word:
            return False
    return True
```

我们不再遍历单词 `word` 中的字母，而是循环遍历必需的单词列表 `required` 。如果单词列表中有任意字母没有出现在单词中，我们可以返回 `False` 。

如果你真的像计算机科学家那样思考的话，应该已经发现，`uses_all` 实际上是已经解决的问题的一个特例，并且可以这么写：

```
def uses_all(word, required):
    return uses_only(required, word)
```



这是被称为将问题回归到已解决问题（reduction to a previously solved problem）的程序开发计划的一个例子。意即你需要识别出的当前问题是一个已经解决的问题的特例，从而可以直接利用现有的解决方案。

## 9.4 使用下标循环

在前面一节例子中，我使用**for** 循环进行遍历，因为只需要字符串中的字符，而不需要操作下标。

但对**is\_abecedarian** 函数我们需要比较相邻的字母，使用**for** 循环比较困难：

```
def is_abecedarian(word):
    previous = word[0]
    for c in word:
        if c < previous:
            return False
        previous = c
    return True
```

或者也可以使用递归：

```
def is_abecedarian(word):
    if len(word) <= 1:
        return True
    if word[0] > word[1]:
        return False
    return is_abecedarian(word[1:])
```

还有一个办法是使用**while** 循环：

```
def is_abecedarian(word):
    i = 0
    while i < len(word)-1:
        if word[i+1] < word[i]:
            return False
        i = i+1
    return True
```

循环开始于*i=0*，并结束于*i=len(word)-1*。每次迭代时，比较第*i*个字符（可以看成是当前字符）和第*i+1*个字符（可以看成是下一个字符）。

如果下一个字符比当前字符小（即按照字母顺序在前），则我们发现了一个破坏字母顺序的断点，可以返回**False**。

如果我们没有找到任何断点而结束循环，则这个单词通过了测试。为了说服自己循环是正确结束的，可以考虑像'**flossy**'这样的例子。这个单词的长度是6，所以最后一次循环时*i*是4，即是倒数第二个字符的下标。在最后一个循环中，会比较倒数第二个和最后一个字符，这正是我们所期待的。

下面是**is\_palindrome** 函数（参考练习 6-3）的一个版本，它使用两个下标；一个从0开始递增；另一个从最后开始递减。

```
def is_palindrome(word):
    i = 0
    j = len(word)-1
    while i<j:
        if word[i] != word[j]:
            return False
        i = i+1
        j = j-1
```

```
return True
```

或者，我们可以将其回归到已经解决的问题，可能这么写：

```
def is_palindrome(word):  
    return is_reverse(word, word)
```

使用练习8-2中的`is_reverse`。

## 9.5 调试

测试程序很难。本章中的函数相对容易测试，因为可以简单地手动验证结果。即便如此，要选择一组可以测试到所有可能的错误的单词，也是很困难的，甚至是不可能的。

举`has_no_e`作为例子，有两个很明显的用例可以检测：包含“e”的单词应该返回`False`；不包含“e”的应当返回`True`。为这两种情况找到具体的单词没有问题。

但对每种情况来说，也存在一些不那么明显的具体情况。在所有包含“e”的单词中，你应当测试以“e”开头的单词，也应当测试以“e”结尾，以及“e”在单词中部的情况。你应当测试长单词、短单词及非常短的单词，如空字符串。空字符串是**特殊情形**（special case）的一个例子。特殊情形往往不那么明显，但又常常隐藏着错误。

除了自己生成的测试用例之外，还可以使用类似`words.txt`这样的单词表来测试你的程序。通过扫描输出，可能会发现错误，但请

注意：你可能发现一种类型的错误（不应该被包含但却被包含的单词），但对另一种类型的则不能发现（应该被包含，但却没有出现的单词）。

总之，测试可以帮助你发现bug，但生成一组好的测试用例并不容易。而且，即使有好的测试用例，也无法确定程序是完全正确的。引用一个传奇计算机科学家的话：

程序测试可以用来显示bug的存在，但无法显示它们的缺席！

(Program testing can be used to show the presence of bugs, but never to show their absence!)

——Edsger W.Dijkstra

## 9.6 术语表

文件对象（file object）：用来表示一个打开的文件的值。

将问题回归到已解决问题（reduction to a previously solved problem）：通过把问题表述为已经解决的某个问题的特例解决问题的一种方式。

特殊情形（special case）：一种不典型或者不明显（因此更可能没有正确处理）的测试用例。

## 9.7 练习

## 练习9-7

本练习中的问题是基于广播节目《车迷天下》（Car Talk）中出现的一个谜题而设计的（<http://www.cartalk.com/content/puzzlers>）：

给我一个包含3组连续的成对字母的单词。我会给你几个几乎可以达到要求却还差一点儿的词作为例子。例如，单词committee，即c-o-m-m-i-t-t-e-e。除了i不满足条件外，这个单词是一个好例子。或者Mississippi: M-i-s-s-i-s-s-i-p-p-i。如果你能够拿掉其中的i，则它也符合要求。但确实有这么一个单词，并且就我所知，它可能是满足这个条件的唯一的单词。当然也可能存在500个，但我只能想到一个。它是什么呢？

编写一个程序来找到它。解答：

<http://thinkpython2.com/code/cartalk1.py>。

## 练习9-8

下面是另一个《车迷天下》中的谜题（<http://www.cartalk.com/content/puzzlers>）：

“有一天我正在高速公路上开车，碰巧注意到里程表。和大部分里程表一样，它显示6位整数的英里数。所以，例如我的车有300 000英里里程，则会看到3-0-0-0-0-0。

“那天我看到的里程数很有意思。我发现最后4位数是回文的；也就是说，它们不论是正序还是逆序地看都一样。例如，5-4-4-5是一个回文，所以我的里程表可能显示为3-1-5-4-4-5。

“1英里之后，后5位数组成一个回文。例如，它可以是3-6-5-4-5-6。再过1英里，6位数的中间4位是一个回文。而接下来，你准备好了吗？又1英里过去，所有的6位数都成了回文！

“问题是，我第一次看里程表时，它的示数是多少？”

编写一个Python程序，检测全部的6位数，并打印出可以满足上面这些要求的数字。解答：<http://thinkpython2.com/code/cartalk2.py>。

### 练习9-9

下面是另一个《车迷天下》的谜题，你可以使用一个搜索来解决（<http://www.cartalk.com/content/puzzlers>）：

“最近我去母亲家时，我发现自己的年龄的两位数正好是母亲的年龄的两位数的倒序。例如，如果她是73岁，我是37岁。我们好奇这种事情这些年来发生过几次，但很快我们的话题就偏转到其他地方，所以没有得到答案。

“我回家后，发现我们的年龄互为倒序的事情至今为止发生过6次。我还发现，如果顺利的话接下来几年还会再遇到一次，并且在那之后如果我们真的很幸运，还能再遇到一次。换句话说，它总共可能发生8次。所以问题是，我现在年龄多大？”

编写一个Python程序，为这个谜题搜索答案。提示：你可能会发现字符串方法`zfill`有用。

解答：<http://thinkpython2.com/code/cartalk3.py>。

---

---

[1] 作者在上一段话中模仿了Gadsby的风格，不使用字母“e”，所以说  
话的风格很怪。因此到这里，他就说“All right, I’ll stop now”，意思是  
停止这种怪异风格的描述。但这个意思无法在译文中表达。上一段话  
的英文原文是：“In fact, it is difficult to construct a solitary thought  
without using that most common symbol. It is slow at first, but with caution  
and hours of training you can gradually gain facility.”——译者注

## 第10章 列表

本章介绍Python语言最有用的内置类型之一：列表。你还能学到更多关于对象的知识，以及同一个对象有两个或更多变量时会发生什么。

### 10.1 列表是一个序列

和字符串相似，**列表**（list）是值的序列。在字符串中，这些值是字符；在列表中，它可以是任何类型。列表中的值称为**元素**（element），有时也称为**列表项**（item）。

创建一个列表有好几种方式。其中最简单的方式是使用方括号（[与 ]）将元素括起来。

```
[10, 20, 30, 40]  
['crunchy frog', 'ram bladder', 'lark vomit']
```

第一个例子是4个整数的列表。第二个例子是3个字符串的列表。列表中的元素并不一定非得是同一类型的。下面的列表包含了一个字符串、一个浮点数、一个整数及（瞧！）另一个列表：

```
['spam', 2.0, 5, [10, 20]]
```

列表中出现的列表是**嵌套的**（nested）。



不包含任何元素的列表称为空列表，可以使用空方括号[] 来创建空列表。

如你所预料的，列表可以赋值给变量：

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
```

## 10.2 列表是可变的

访问列表元素的语法和访问字符串中字符的语法是一样的——使用方括号操作符。方括号中的表达式指定下标。请记住下标是从0开始的：

```
>>> cheeses[0]
'Cheddar'
```

和字符串不同的是，列表是可变的。当方括号操作符出现在赋值语句的左侧时，它用于指定列表中哪个元素会被赋值。

```
>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

`numbers` 的第1位元素，原先的值是123，现在是5了。

图10-1显示了`cheeses`、`numbers` 和`empty` 的状态图。

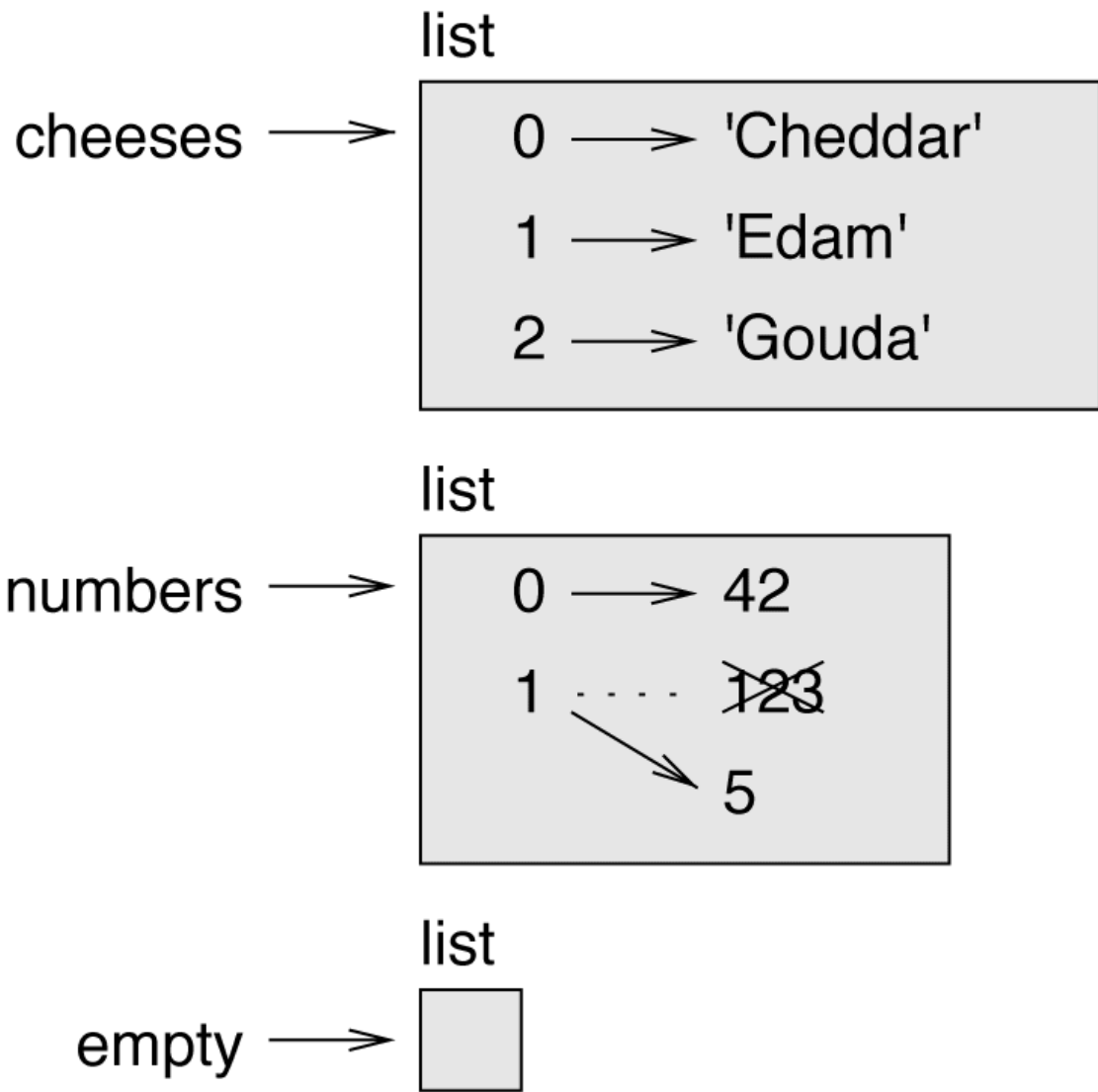


图10-1 状态图

在图10-1中，外面写有“list”的图框表示列表，里面显示的是列表中的元素。**cheeses** 变量引用着一个列表，包含3个元素，下标分别是0、1和2。**numbers** 包含两个元素；本图显示了其第二个元素从123重新赋值为5的过程。**empty** 引用一个没有任何元素的空列表。

列表下标和字符串下标工作方式相同。

- 任何整型的表达式都可以用作下标。
- 如果尝试读写一个并不存在的元素，则会得到`IndexError`。
- 如果下标是负数，则从列表的结尾处反过来数下标访问。

`in` 操作符也可以用于列表。

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

## 10.3 遍历一个列表

遍历一个列表元素的最常见方式是使用`for` 循环。语法和字符串的遍历相同：

```
for cheese in cheeses:
    print(cheese)
```

在只需要读取列表的元素本身时，这样的遍历方式很好。但如果需要写入或者更新元素时，则需要下标。一个常见的方式是使用内置函数`range` 和`len`：

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

这个循环遍历列表，并更新每个元素。`len` 返回列表中元素的个数。`range` 返回一个下标的列表，从0到 $n-1$ ，其中 $n$  是列表的长度。

每次迭代时，**i** 获得下一个元素的下标。循环体中的赋值语句使用**i** 来读取元素的旧值并赋值为新值。

在空列表上使用**for** 循环，则循环体从不会被运行：

```
for x in []:  
    print('This never happens.')
```

虽然列表可以包含其他的列表，嵌套的列表仍然被看作一个单独的元素。下面的列表长度是4：

```
['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

## 10.4 列表操作

+操作符可以拼接列表：

```
>>> a = [1, 2, 3]  
>>> b = [4, 5, 6]  
>>> c = a + b  
>>> c  
[1, 2, 3, 4, 5, 6]
```

\*操作符重复一个列表多次：

```
>>> [0] * 4  
[0, 0, 0, 0]  
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

第一个例子重复列表[0] 四次。第二个例子重复列表[1, 2, 3] 三次。

## 10.5 列表切片

切片操作符也可以用于列表：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

如果省略掉第一个下标，则切片从列表开头开始。如果省略掉第二个下标，则切片至列表结尾结束。如果两个下标都省略，则切片就是整个列表的副本。

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

因为列表是可变的，所以在对列表进行修改操作之前，复制一份是很有用的。

如果切片操作符出现在赋值语句的左侧，则可以更新多个元素：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
['a', 'x', 'y', 'd', 'e', 'f']
```

## 10.6 列表方法

Python为列表提供了不少操作方法。例如，**append** 可以在列表尾部添加新的元素：

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

**extend** 方法接收一个列表作为参数，并将其所有的元素附加到列表中：

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

这个例子中**t2** 没有被修改。

**sort** 方法将列表中的元素从低到高重新排列：

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

列表的大多数方法全是无返回值的。它们修改列表，并返回**None**。如果不小心写了**t = t.sort()**，你可能对结果感到很失望。

## 10.7 映射、过滤和化简

如果想把列表中所有的元素加起来，可以使用下面这样的循环：

```
def add_all(t):  
    total = 0  
    for x in t:  
        total += x  
    return total
```

`total` 被初始化为0。每次循环中，`x` 获取列表中的一个元素。`+=` 操作符为更新变量提供了一个简洁的方式。这个**增加赋值语句**：

```
total += x
```

等价于：

```
total = total + x
```

随着循环的运行，`total` 会累积列表中的值的和；这样使用一个变量有时称为**累加器**（accumulator）。

对列表元素累加是如此常见的操作，以至于Python提供了一个内置函数`sum`：

```
>>> t = [1, 2, 3]  
>>> sum(t)  
6
```

类似这样，将一个序列的元素值合起来到一个单独的变量的操作，有时称为**化简**（reduce）。

有时候你想要在遍历一个列表的同时构建另一个列表。例如，下面的函数接收一个字符串列表，并返回一个新列表，其元素是大写的字符串：

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

`res` 初始化为一个空列表；每次循环，我们给它附加一个元素。所以`res` 也是一种累加器。

像`capitalize_all` 这样的操作，有时被称为**映射**（`map`），因为它将一个函数（在这个例子里是`capitalize` 方法）“映射”到一个序列的每个元素上。

另一个常见的操作是选择列表中的某些元素，并返回一个子列表。例如，下面的函数接收一个字符串列表，并返回那些只包含大写字母的字符串：

```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

`isupper` 是一个字符串方法，当字符串中只包含大写字母时返回 `True` 。



类似`only_upper` 这样的操作称为**过滤**（`filter`），因为它选择列表中的某些元素，并过滤掉其他的元素。

列表的绝大多数常用操作都可以用映射、过滤和化简的组合来表达。

## 10.8 删除元素

从列表中删除元素，有多种方法。如果知道元素的下标，可以使用`pop`：

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

`pop` 修改列表，并返回被删除掉的值。如果不提供下标，它会删除并返回最后一个元素。

如果不需要使用删除的值，可以使用`del` 操作符：

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> t
['a', 'c']
```

如果知道要删除的元素（而不是下标），则可以使用`remove`：

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
```

```
['a', 'c']
```

`remove` 方法的返回值是`None`。

若要删除多个元素，可以使用`del` 和切片下标：

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
['a', 'f']
```

和通常一样，切片会选择所有的元素，直到第二个下标（并不包含）。

## 10.9 列表和字符串

字符串是字符的序列，而列表是值的序列，但字符的列表和字符串并不相同。若要将一个字符串转换为一个字符的列表，可以使用函数`list`：

```
>>> s = 'spam'
>>> t = list(s)
>>> t
['s', 'p', 'a', 'm']
```

由于`list` 是内置函数的名称，所以应当尽量避免使用它作为变量名称。我也避免使用`l`，因为它看起来太像数字`1`了。因而我使用`t`。

`list` 函数会将字符串拆成单个的字母。如果想要将字符串拆成单词，可以使用`split` 方法：

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> t
['pining', 'for', 'the', 'fjords']
```

`split` 还接收一个可选的形参，称为**分隔符**（`delimiter`），用于指定用哪个字符来分隔单词。下面的例子中使用连字符（`-`）作为分隔符：

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
['spam', 'spam', 'spam']
```

`join` 是 `split` 的逆操作。它接收字符串列表，并拼接每个元素。`join` 是字符串的方法，所以必须在分隔符上调用它，并传入列表作为实参：

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
'pining for the fjords'
```

在这个例子里，分隔符是空格，所以 `join` 会在每个单词之间放一个空格。若想不用空格直接连接字符串，可以使用空字符串 `''` 作为分隔符。

## 10.10 对象和值

如果我们运行下面的赋值语句：

```
a = 'banana'
b = 'banana'
```

我们知道**a**和**b**都是一个字符串的引用。但我们不知道它们是否指向同一个字符串。有两种可能的状态，如图10-2所示。



图10-2 状态图

一种可能是，**a**和**b**引用着不同的对象，它们的值相同。另一种情况下，它们指向同一个对象。

要检查两个变量是否引用同一个对象，可以使用**is**操作符。

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

在这个例子里，Python只建立了一个字符串对象，而**a**和**b**都引用它。

但当你新建两个列表时，会得到两个对象：

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

所以状态图如图10-3所示。

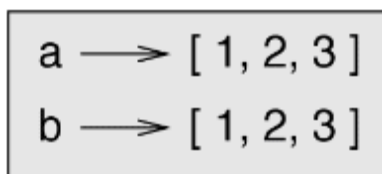


图10-3 状态图

在这个例子里我们会说这两个列表是**相等的**（equivalent），因为它们有相同的元素，但它们不是**相同的**（identical），因为它们并不是同一个对象。如果两个对象相同，则必然也相等，但如果两个对象相等，并不一定相同。

到目前为止，我们都不加区分地使用“对象”和“值”，但更精确的说法是对象有一个值。如果求值`[1, 2, 3]`，会得到一个列表对象，它的值是一个整数的序列。如果另一个列表包含相同的元素，我们说它有相同的值，但它们不是同一个对象。

## 10.11 别名

如果`a`引用一个对象，而你赋值`b = a`，则两个变量都会引用同一个对象：

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

这里的状态图如图10-4所示。

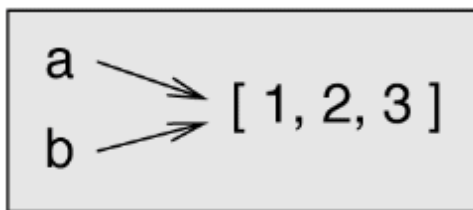


图10-4 状态图

变量和对象之间的关联关系称为**引用**（reference）。在这个例子里，有两个指向同一对象的引用。

当一个对象有多个引用，并且引用有不同的名称时，我们说这个对象有**别名**（aliased）。

如果有别名的对象是可变的，则对一个别名的修改会影响另一个：

```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

虽然这种行为可能很有用，但它也容易导致错误。通常来说，当处理可变对象时，避免使用别名会更加安全。

对于字符串这样的不可变对象，别名则不会带来问题。在下面的例子中：

```
a = 'banana'
b = 'banana'
```

不论**a** 和**b** 是否引用同一个字符串，都不会有什么区别。

## 10.12 列表参数

当你将一个列表传递给函数中，函数会得到列表的一个引用。如果函数中修改了列表，则调用者也会看到这个修改。例如，`delete_head` 函数删除列表中的第一个元素：

```
def delete_head(t):  
    del t[0]
```

下面使用它：

```
>>> letters = ['a', 'b', 'c']  
>>> delete_head(letters)  
>>> letters  
['b', 'c']
```

参数`t` 和变量`letters` 是同一个对象的别名。栈图如图 10-5所示。

因为列表被两个帧共享，所以我将它画在中间。

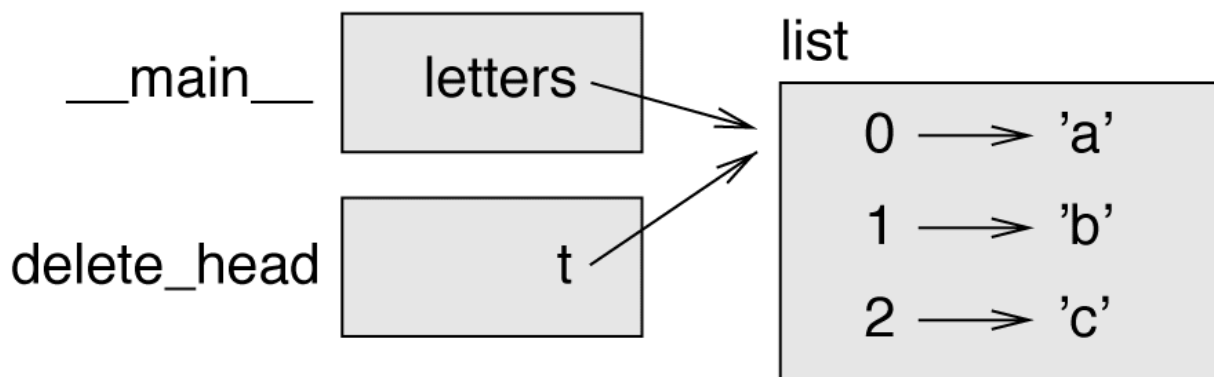


图10-5 栈图

区分修改列表的操作和新建列表的操作十分重要。例如，`append` 方法修改列表，但是`+` 操作符新建一个列表：

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
```

`append` 修改列表，返回`None`。

```
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
>>> t1
```

操作符`+`创建一个新列表，而原始的列表并不改变。

这个区别，在编写希望修改列表的函数时十分重要。例如，下面的函数并不会删除列表的开头：

```
def bad_delete_head(t):
    t = t[1:]                                # 错!
```

切片操作会新建一个列表，而赋值操作会让 `t` 引用指向这个新的列表，但这些操作对调用者没有影响。

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```



在`bad_delete_head`的开头，`t`和`t4`指向同一个列表。在函数最后，`t`指向了一个新的列表，但`t4`仍然指向原先的那个没有改变的列表。

另外一种方法是编写函数创建和返回一个新的列表。例如，`tail`返回除了第一个以外所有的元素的列表：

```
def tail(t):  
    return t[1:]
```

这个函数不会修改原始列表。下面的代码展示如何使用它：

```
>>> letters = ['a', 'b', 'c']  
>>> rest = tail(letters)  
>>> rest  
['b', 'c']
```

## 10.13 调试

对列表（以及其他可变对象）的不慎使用，可能会导致长时间的调试。下面介绍一些常见的陷阱，以及如何避免它们。

1. 大部分列表方法都是修改参数并返回`None`的。这和字符串的方法正相反，字符串方法新建一个字符串，并留着原始的字符串不动。

如果你习惯于写下面这样的字符串代码：

```
word = word.strip()
```

则容易倾向于这么写列表代码：

```
t = t.sort()           # 错！
```

因为`sort` 返回`None`，接下来对`t` 进行的操作很可能会失败。

在使用列表方法和操作符之前，应当仔细阅读文档，并在交互模式中测试它们。

## 2. 选择一种风格，并坚持不变。

列表的问题之一是同样的事情有太多种可用的做法。例如，要从列表中删除一个元素，可以使用`pop`、`remove`、`del` 或者甚至是切片赋值。

要添加一个元素，可以使用`append` 方法或者`+` 操作符。假设`t` 是一个列表，`x` 是一个列表元素，下面的操作是正确的：

```
t.append(x)
t = t + [x]
t += [x]
```

而下面的操作是错误的：

```
t.append([x])          # 错！
t = t.append(x)         # 错！
t + [x]                 # 错！
t = t + x               # 错！
```

在交互模式中试验这些例子，确保你明白它们的运行细节。注意只有最后一个会导致运行时错误；其他的3个都是合法的，但是它们的结果不正确。

### 3. 通过复制来避免别名。

如果想要使用类似**sort** 的方法来修改参数，但又需要保留原先的列表，可以复制一个副本：

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

在这个例子里也可以使用内置函数**sorted**，它会返回一个新的排好序的列表，并且留着原先的列表不动。

```
>>> t2 = sorted(t)
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

## 10.14 术语表

列表（**list**）：值的序列。

元素（**element**）：列表（或其他序列）中的一个值，也称为列表项。

嵌套列表（**nested list**）：作为其他列表的元素的列表。

累加器（**accumulator**）：在循环中用于加和或者累积某个结果的变量。

增加赋值（**augmented assignment**）：使用类似+=操作符来更新变量值的语句。

化简（**reduce**）：一种处理模式，遍历一个序列，并将元素的值累积起来计算为一个单独的结果。

映射（**map**）：一种处理模式，遍历一个序列，对每个元素进行操作。

过滤（**filter**）：一种处理模式，遍历列表，并选择满足某种条件的元素。

对象（**object**）：变量可以引用的东西。对象有类型和值。

相等（**equivalent**）：拥有相同的值。

相同（**identical**）：是同一个对象（并且也意味着相等）。

引用（**reference**）：变量和它的值之间的关联。

别名（**aliasing**）：多个变量同时引用一个对象的情况。

分隔符（**delimiter**）：用于分隔字符串的一个字符或字符串。

## 10.15 练习

你可以从[http://thinkpython2.com/code/list\\_exercises.py](http://thinkpython2.com/code/list_exercises.py)下载这些练习的解答。

### 练习10-1

编写一个名为**nested\_sum**的函数，接收一个由内嵌的整数列表组成的列表作为形参，并将内嵌列表中的值全部加起来。例如：

```
>>> t = [[1, 2], [3], [4, 5, 6]]
>>> nested_sum(t)
21
```

### 练习10-2

编写一个名为**cumsum**的函数，接收一个数字的列表，返回累计和；也就是说，返回一个新的列表，其中第*i*个元素是原先列表的前*i*+1个元素的和。例如：

```
>>> t = [1, 2, 3]
>>> cumsum(t)
[1, 3, 6]
```

### 练习10-3

编写一个函数**middle**，接收一个列表作为形参，并返回一个新列表，包含除了第一个和最后一个元素之外的所有元素。例如：

```
>>> t = [1, 2, 3, 4]
>>> middle(t)
[2, 3]
```

### 练习10-4

编写一个名为**chop**的函数，接收一个列表，修改它，删除它的第一个和最后一个元素，并返回**None**。例如：

```
>>> t = [1, 2, 3, 4]
>>> chop(t)
>>> t
[2, 3]
```

### 练习10-5

编写一个名为**is\_sorted**的函数，接收一个列表作为形参，并当列表是按照升序排好序的时候返回**True**，否则返回**False**。

例如：

```
>>> is_sorted([1, 2, 2])
True
>>> is_sorted(['b', 'a'])
False
```

### 练习10-6

两个单词，如果重新排列其中一个的字母可以得到另一个，它们互为回文（**anagram**）。编写一个名为**is\_anagram**的函数，接收两个字符串，当它们互为回文时返回**True**。

### 练习10-7

编写一个名为`has_duplicates`的函数接收一个列表，当其中任何一个元素出现多于一次时返回`True`。它不应当修改原始列表。

### 练习10-8

这个练习谈的是所谓的生日悖论，你可以在[http://en.wikipedia.org/wiki/Birthday\\_paradox](http://en.wikipedia.org/wiki/Birthday_paradox)阅读相关资料。

如果你的班级中有23个学生，那么其中有两人生日相同的概率有多大？你可以通过随机生成23个生日的样本并检查是否有相同的匹配来估计这个概率。提示：可以使用`random`模块中的`randint`函数来生成随机生日。

你可以从<http://thinkpython2.com/code/birthday.py>下载解答。

### 练习10-9

编写一个函数，读取文件`words.txt`，并构建一个列表，每个元素是一个单词。给这个函数编写两个版本，其中一个使用`append`方法，另一个使用`t = t + [x]`的用法。哪一个运行时间更长？为什么？

解答：<http://thinkpython2.com/code/wordlist.py>。

### 练习10-10

要检查一个单词是否出现在单词列表中，可以使用`in`操作符，但由于它需要按顺序搜索所有单词，可能会比较慢。

因为单词是按字母顺序排列的，我们可以使用二分查找（也叫作二分搜索）来加快速度。二分查找的过程类似于在字典中查找单词。从中间开始，检查需要找的单词是不是在列表中间出现的单词之前，如果是，则继续用同样的方法搜索前半部分。否则搜索后半部分。

不论哪种情形，都将搜索空间减小了一半。如果单词列表有113,809个单词，那么大概耗费17步就能找到单词，或者确认它不在列表之中。

编写一个函数**in\_bisect**，接收一个排好序的列表，以及一个目标值，当目标值在列表之中，返回其下标，否则返回**None**。

或者你可以阅读**bisect** 模块的文档，并使用它！

解答：<http://thinkpython.com2/code/inlist.py>。

### 练习10-11

两个单词，如果其中一个是另一个的反向序列，则称它们为“反向对”。编写一个程序找到单词表中出现的全部反向对。

解答：[http://thinkpython2.com/code/reverse\\_pair.py](http://thinkpython2.com/code/reverse_pair.py)。

### 练习10-12

两个单词，如果从每个单词中交错取出一个字母可以组成一个新的单词，我们称它们为“互锁”（interlocking）。例如，“shoe”和“cold”可以互锁组成单词“schooled”。



解答: <http://thinkpython2.com/code/interlock.py>。鸣谢: 这个练习启发自<http://puzzlers.org>的一个示例。

1. 编写一个程序找到所有互锁的词。提示: 不要穷举所有的词对!

2. 能不能找到“三互锁”的单词? 也就是, 从第一、第二或者第三个字母开始, 每第三个字母合起来可以形成一个单词。

## 第11章 字典

本章介绍另一种内置类型：字典。字典是Python最好的语言特性之一，它是很多高效而优雅的算法的基本构建块。

### 11.1 字典是一种映射

**字典** 类似于列表，但更加通用。在列表中，下标必须是整数；而在字典中，下标（几乎）可以是任意类型。

字典包含下标（称为**键**）集合和值集合。每个键都与一个值关联。键和值之间的关联被称为**键值对**（key-value pair），或者有时称为**一项**（item）。

用数学语言来描述，字典体现了键到值的**映射**，所以可以说每个键“映射”到一个值。作为示例，我们构建一个字典，将英语单词映射到西班牙语上，所以键和值的类型都是字符串。

函数**dict** 新建一个不包含任何项的字典。因为**dict** 是内置函数的名称，应当避免使用它作为变量名。

```
>>> eng2sp = dict()
>>> eng2sp
{}
```

这里花括号`{}`表示一个空的字典。想要给字典添加新项，可以使用方括号操作符：

```
>>> eng2sp['one'] = 'uno'
```

这一行代码创建一个新项，将键`'one'`映射到值`'uno'`上。如果我们再次打印这个字典，可以看到一个键值对，以冒号分隔：

```
>>> eng2sp  
{ 'one': 'uno' }
```

这种输出格式也同样是输入的格式。例如，可以创建一个包含3项的新字典：

```
>>> eng2sp = { 'one': 'uno', 'two': 'dos', 'three': 'tres' }
```

但如果你打印`eng2sp`，可能会觉得奇怪：

```
>>> eng2sp  
{ 'one': 'uno', 'three': 'tres', 'two': 'dos' }
```

字典中键值对的顺序可能并不相同。如果你在自己的电脑上输入相同的示例，可能会得到另一个不同的结果。总之，字典中各项的顺序是不可预料的。

但这并不是问题，因为字典的元素从来不使用整数下标进行查找。相对地，它使用键来查找对应的值：

```
>>> eng2sp['two']  
'dos'
```

如果键 `'two'` 总是映射到值 `'dos'` 上，那么各项的顺序其实并不重要。

如果一个键并不在字典之中，会得到一个异常：

```
>>> eng2sp['four']
KeyError: 'four'
```

`len` 函数可以用在字典上，它返回键值对的数量：

```
>>> len(eng2sp)
3
```

`in` 操作符也可以用在字典上，它告诉你一个值是不是字典中的键（是字典中的值则不算）。

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

若要查看一个值是不是出现在字典的值中，可以使用方法 `values`，它会返回一个值集合，并可以应用 `in` 操作符：

```
>>> vals = eng2sp.values()
>>> 'uno' in vals
True
```

`in` 操作符对列表和字典使用不同的算法实现。对于列表，它按顺序搜索列表的元素，如8.6节所示。当列表变长时，搜索时间会随之变长。

而对于字典，Python使用一个称为**散列表**（hashtable）的算法。它有一个值得注意的特点：不管字典中有多少项，`in` 操作符花费的时间都差不多。我会在21.4节中解释其中的原因，但最好再多读几章，这样才可能看懂解释的内容。

## 11.2 使用字典作为计数器集合

假设给定一个字符串，你想要计算每个字母出现的次数。有几种可能的实现方法：

1. 你可以创建26个变量，每个变量对应字母表上的一个字母。接着遍历字符串，对每一个字符，增加对应的计数器。你可能需要使用一个链式条件判断。

2. 你可以创建一个包含26个元素的列表。接着可以将每个字符转换为一个数字（使用内置函数`ord`），使用这个数字作为列表的下标，并增加对应的计数器。

3. 你可以建立一个字典，以字符作为键，以计数器作为相应的值。第一次遇到某个字符时，在字典中添加对应的项。之后可以增加一个已经存在的项的值。

这几种方案进行相同的计算，但实现计算的方式不一样。

**实现**（implementation）是进行某种计算的一个具体方式；有的实现比其他的更好。例如，字典实现的优势之一是我们并不需要预先知道字符串中可能出现哪些字母，因而只需为真正出现过的字母分配空间。

下面是这个实现的代码：

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

这个函数的名称是**直方图**（histogram），它是一个统计学术语，表示一个计数器（或者说频率）的集合。

函数的第一行创建一个空的字典。**for** 循环遍历字符串。每次迭代中，如果字符**c** 不在字典中，我们就创建一个新项，其键是**c**，其值初始化为1（因为我们已经见到这个字符一次了）。如果**c** 已经在字典之中，我们增加**d[c]**。

下面是这个函数的使用方式：

```
>>> h = histogram('brontosaurus')
>>> h
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

这个直方图显示，字母'**a**' 和 '**b**' 出现了1次； '**o**' 出现了两次，依此类推。

字典有一个方法`get`，接收一个键以及一个默认值。如果键出现在字典中，`get` 返回对应的值；否则它返回默认值。例如：

```
>>> h = histogram('a')
>>> h
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

作为练习，使用`get` 将`histogram` 写得更紧凑一些。你应当可以消除掉`if` 语句。

## 11.3 循环和字典

如果在`for` 循环中使用字典，会遍历字典的键。例如，`print_hist` 函数打印字典的每一个键以及对应的值：

```
def print_hist(h):
    for c in h:
        print(c, h[c])
```

下面是这个函数输出的样子：

```
>>> h = histogram('parrot')
>>> print_hist(h)
a 1
p 1
r 2
t 1
o 1
```

同样地，键的出现没有特定的顺序。要按顺序遍历所有键，可以使用内置函数`sorted`：

```
>>> for key in sorted(h):
...     print(key, h[key])
a 1
o 1
p 1
r 2
t 1
```

## 11.4 反向查找

给定一个字典`d`和键`k`，找到对应的值`v = d[k]`非常容易。这个操作称为**查找**（lookup）。

但是如果有`v`，而想找到`k`时怎么办？这里有两个问题：首先，可能存在多个键映射到同一个值`v`上。随不同的应用场景，也许可以挑其中一个，或者也许需要建立一个列表来保存所有的键。其次，并没有可以进行**反向查找**的简单语法，你需要使用搜索。

下面是一个函数，接收一个值，并返回映射到该值的第一个键：

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise LookupError()
```

这个函数是搜索模式的又一个示例。但它使用了一个我们还没见过的语言特性，`raise`语句。**raise 语句**会生成一个异常；在这个



例子里它生成一个**LookupError**，这是一个内置异常，通常用来表示查找操作失败。

如果我们到达了循环的结尾，就意味着**v** 在字典中没有作为值出现过，所以我们抛出一个异常。

下面的例子展示了一个成功的反向查找：

```
>>> h = histogram('parrot')
>>> k = reverse_lookup(h, 2)
>>> k
'r'
```

以及一个不成功的反向查找：

```
>>> k = reverse_lookup(h, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in reverse_lookup
LookupError
```

当你自己抛出异常时，效果和**Python**抛出异常是一样的：它会打印出一个回溯和一个错误信息。

**raise** 语句也可以接收一个可选的参数用来详细描述错误。例如：

```
>>> raise LookupError('value does not appear in the dictionary')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
LookupError: value does not appear in the dictionary
```

反向查找远远慢于正向查找；如果频繁这么做，或者字典非常大时，会对程序的性能有很大影响。

## 11.5 字典和列表

列表可以在字典中以值的形式出现。例如，如果你遇到一个将字母映射到频率的字典，可能会想要反转它；也就是说，建立一个字典，将频率映射到字母上。因为可能出现多个字母频率相同的情况，在反转的字典中，每项的值应当是字母的列表。

这里是一个反转字典的函数：

```
def invert_dict(d):
    inverse = dict()
    for key in d:
        val = d[key]
        if val not in inverse:
            inverse[val] = [key]
        else:
            inverse[val].append(key)
    return inverse
```

每次循环中，**key** 从 **d** 中获得一个键，而 **val** 获得相应的值。如果 **val** 不在 **inverse** 字典中，意味着我们还没有见到过它，所以新建一个项，并将它初始化为一个**单件**（**singleton**，即只包含一个元素的列表）。否则我们已经见过这个值了，因此将相应的键附加到列表末尾。

下面是一个示例：

```
>>> hist = histogram('parrot')
>>> hist
```

```
{'a': 1, 'p': 1, 'r': 2, 't': 1, 'o': 1}
>>> inverse = invert_dict(hist)
>>> inverse
{1: ['a', 'p', 't', 'o'], 2: ['r']}
```

图11-1是显示`hist` 和`inverse` 的状态图。字典使用一个上方标明`dict` 的图框表示，内部包含键值对。如果值是整数、浮点数或字符串，我会把它们画到图框内，但我常常会将列表画在图框之外，以便保持状态图的简洁。

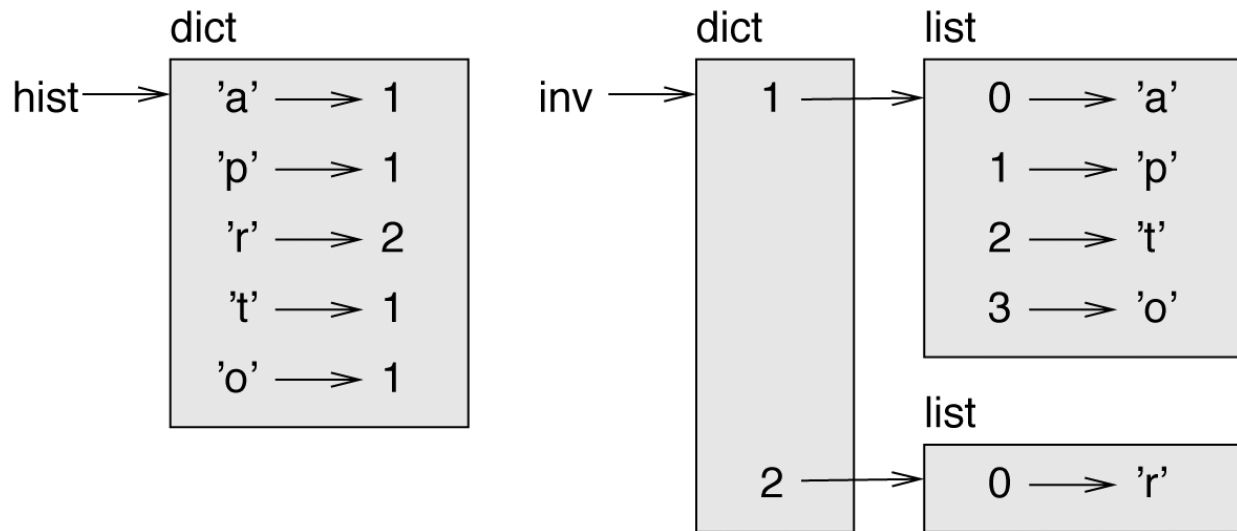


图11-1 状态图

如本例所示，列表可以用作字典的值，但它们不能用作键。如果尝试的话，会得到如下的结果：

```
>>> t = [1, 2, 3]
>>> d = dict()
>>> d[t] = 'oops'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
```

之前我提到过字典是通过散列表的方式实现的，这意味着键必须是**可散列**（hashable）的。

**散列**是一个函数，接收（任意类型）的值并返回一个整数。字典使用这些被称为散列值的整数来保存和查找键值对。

这套系统当键不可变时，可以正确工作。但如果像列表这样，键是可变的话，则会有不好的事情发生。例如，新建一个键值对时，Python将键进行散列并存储到对应的地方。如果修改了键并再次散列，它会指向一个不同的地方。在那种情况下，会导致同一个键有两个条目，或者可能找不到某个键。不论如何，字典将无法正确工作。

因此键必须是可散列的，而类似列表这样的可变类型是不可散列的。绕过这种限制的最简单办法是使用元组，下一章会有详细介绍。

因为字典是可变的，它也不能用作键，但它可以用作字典的值。

## 11.6 备忘

如果你尝试过6.7节中的**fibonacci**函数，可能会注意到，提供的参数越大，函数运行的时间越长，并且运行时间增长很快。

为了明白为什么会这样，考虑图11-2，它展示了**fibonacci**函数  $n=4$  时的**调用图**。

调用图显示了一组函数帧，并用箭头将函数的帧和它调用的函数帧连接起来。在图的顶端， $n=4$  的**fibonacci**调用了 $n=3$ 和 $n=2$ 的

`fibonacci`。同样地，`n=3` 的 `fibonacci` 调用了 `n=2` 和 `n=1` 的 `fibonacci`。依此类推。

数一下 `fibonacci(0)` 和 `fibonacci(1)` 被调用了多少次。这是本问题的一个很低效的解决方案，而且当参数变大时，事情会变得更糟。

一个解决办法是记录已经计算过的值，并将它们保存在一个字典中。将之前计算的值保存起来以便后面使用的方法称为**备忘**（memo）。下面是一个使用了备忘的 `fibonacci` 版本：

```
known = {0:0, 1:1}

def fibonacci(n):
    if n in known:
        return known[n]

    res = fibonacci(n-1) + fibonacci(n-2)
    known[n] = res
    return res
```

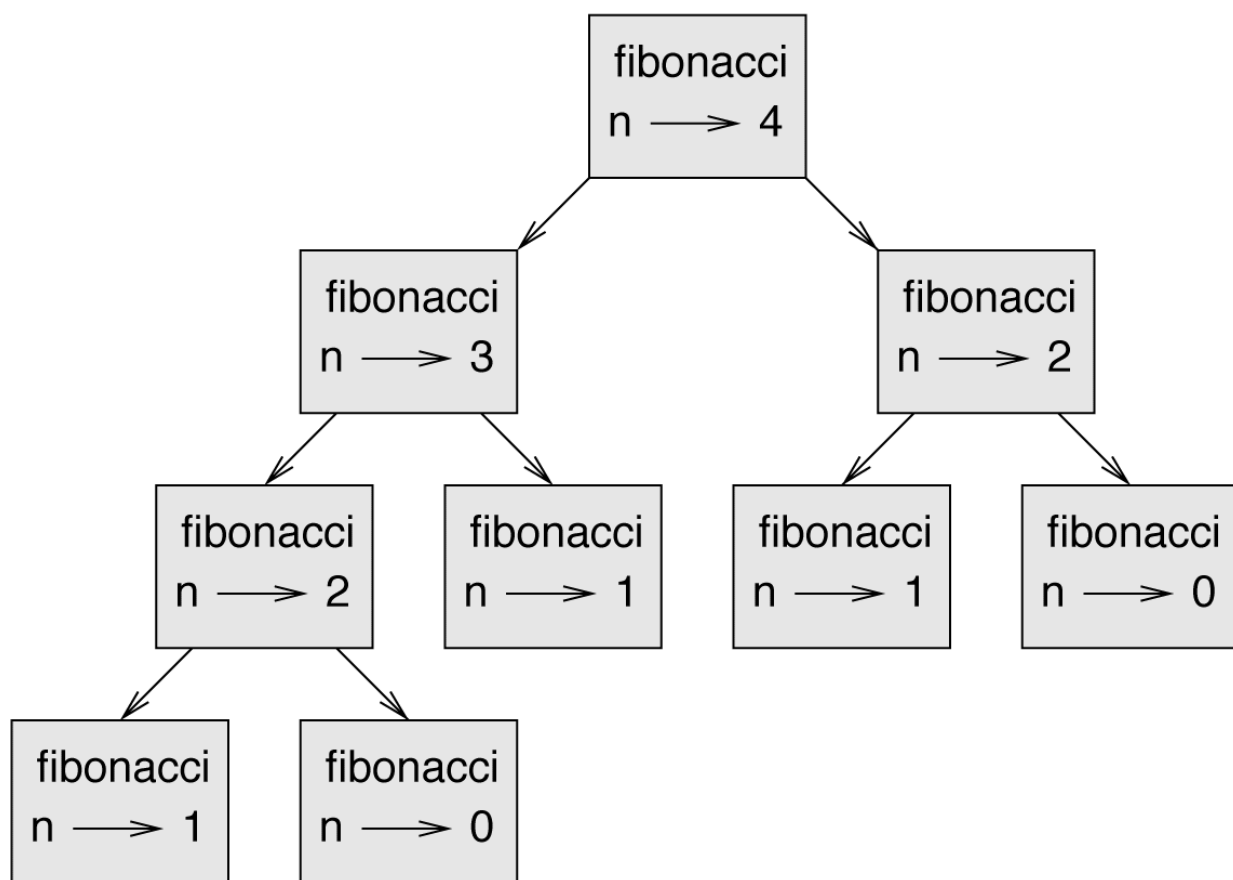


图11-2 调用图

`known` 是一个用来记录我们已知的Fibonacci数的字典。开始时它有两项：0映射到0，以及1映射到1。

每当`fibonacci`被调用时，它会先检查`known`。如果结果已经存在，则可以立即返回。如果不存在，它需要计算这个新值，将其添加进字典，并返回。

如果你运行`fibonacci`的这个版本，并将其与原始版本进行比较，你会发现，这个版本快得多。

## 11.7 全局变量

在前一个例子中，`known` 是在函数之外创建的，所以它属于被称为 `__main__` 的特殊帧。`__main__` 之中的变量有时被称为**全局** 变量，因为它们可以在任意函数中访问。和局部变量在函数结束时就消失不同，全局变量可以在不同函数的调用之间持久存在。

全局变量常常用作**标志**（flag）；它是一种布尔变量，可以标志一个条件是否为真。例如，有的函数使用一个叫 `verbose` 的标志来控制输出的详细程度：

```
verbose = True

def example1():
    if verbose:
        print('Running example1')
```

如果你尝试给全局变量重新赋值，可能会感到惊讶。下面例子的本意是想记录函数是否被调用过：

```
been_called = False

def example2():
    been_called = True    # 错
```

但当你运行它时，会发现 `been_called` 的值并不会变化。问题在于函数 `example2` 会新建一个局部变量 `been_called`。局部变量在函数结束时就会消失，并且对全局变量没有任何影响。

要想在函数中给全局变量重新赋值，你需要在使用它之前先**声明**这个全局变量：

```
been_called = False

def example2():
    global been_called
    been_called = True
```

`global` 语句告诉编译器，“在这个函数里，当我说 `been_called` 时，我指的是全局变量；不要新建一个局部变量。”

下面是一个尝试更新全局变量的例子：

```
count = 0

def example3():
    count = count + 1          # 错
```

如果运行它，会得到：

```
UnboundLocalError: local variable 'count' referenced before
assignment
```

Python 会假设 `count` 是局部的，在这种假设下你在写入它之前先读取了。解决方案也是声明 `count` 为全局变量。

```
def example3():
    global count
    count += 1
```

如果全局变量指向的是可变的值，可以不用声明该变量就可以修改该值：

```
known = {0:0, 1:1}

def example4():
```



```
known[2] = 1
```

所以你可以添加、删除和替换一个全局的列表或字典的元素，但如果想要给全局变量重新赋值，则需要声明它：

```
def example5():  
    global known  
    known = dict()
```

全局变量很有用，但是如果使用太多，并且频繁修改，可能会让代码比较难调试。

## 11.8 调试

在使用更大的数据集时，通过打印和手动检查输出的方式来调试已经变得很笨拙了。下面是一些调试大数据集的建议。

### 缩小输入

如果可能，减小数据集的尺寸。例如，程序如果读入文本文件，可以从开头10行开始，或者使用你能找到的最小样本。你可以编辑文件本身，或者（更好地）修改程序让它只读入前 $n$ 行。

如果出现了错误，可以调小 $n$ ，小到足够展现出错误的最小程度，并在修改之后逐渐增大 $n$ 。

### 检查概要信息和类型

与其打印和检查整个数据集，可以考虑打印出数据的概要信息：例如，字典中条目的数量，或者一个列表中数的和。

运行时错误的一个常见原因是某个值的类型不对。调试这种错误时，常常只需要打印出值的类型就足够了。

## 编写自检查逻辑

有时候可以写代码自动检查错误。例如，如果你要计算一系列数的平均值，可以检查结果是否比列表中最大的数小，或者比最小的数大。这种检查称为“健全检查”（sanity check），因为它会发现那些“发疯”的结果。

另一种检查可以对比两种不同的计算的结果，查看它们是否一致。这样的检查称为“一致性检查”。

## 格式化输出

格式化调试输出，可以更容易发现错误。我们在6.9节中已经看到过一个例子。`pprint` 模块提供了一个 `pprint` 函数，可以将内置类型的值以更加人性化的可读的格式打印出来。（`pprint` 代表“pretty print”。）

另外，再提醒一次，花费时间构建脚手架代码，可以减少未来进行调试的时间。

## 11.9 术语表

映射（mapping）：一个集合中的每个元素与另一个集合中的元素所产生的关联。

字典（dictionary）：从键到对应的值的映射。

键值对（key-value pair）：键到值的映射的展示。

项（item）：在字典中，键值对的另一个名称。

键（key）：字典中出现在键值对的前一部分的对象。

值（value）：字典中出现在键值对的后一部分的对象。这比我们之前提到的“值”更加具体。

实现（implementation）：进行计算的一个具体方式。

散列表（hashtable）：Python字典的实现用的算法。

散列函数（hash function）：散列表中用来计算一个键的位置的函数。

可散列（hashable）：拥有散列函数的类型。不可变类型，诸如整数、浮点数和字符串都是可散列的；可变类型，诸如列表和字典，都是不可散列的。

查找（lookup）：字典的一个操作，接收一个键，并找到它对应的值。

反向查找（**reverse lookup**）：字典的一个操作，通过一个值来找到它对应的一个或多个键。

**raise** 语句（**raise statement**）：一个（故意）抛出异常的语句。

单件（**singleton**）：只包含一个元素的列表（或其他序列）。

调用图（**call graph**）：一个用来展示程序运行中创建的每一帧的关系的图。使用箭头连接每个调用者和被调用者。

备忘（**memo**）：将计算的结果存储起来，以避免将来进行不必要的计算。

全局变量（**global variable**）：在函数之外定义的变量。全局变量可以在任何函数中访问。

全局语句（**global statement**）：声明变量名为全局的语句。

标志（**flag**）：用于标志一个条件是否为真的布尔变量。

声明（**declaration**）：类似于**global**这样的用于通知解释器关于一个变量的信息的语句。

## 11.10 练习

### 练习11-1

编写一个函数，读入**words.txt**中的单词，并将其作为键保存到一个字典中。字典的值是什么并不重要。然后你就可以使用**in**操作符

快速检查一个字符串是否在这个字典中。

如果你做过了练习10-10，可以将这个实现与列表的`in`操作符以及二分查找进行速度的对比。

### 练习11-2

阅读字典方法`setdefault`的文档，并使用它来写一个更简洁的`invert_dict`。

解答：[http://thinkpython2.com/code/invert\\_dict.py](http://thinkpython2.com/code/invert_dict.py)。

### 练习11-3

将练习6-2中的`Ackermann`函数改为备忘化的版本，并查看备忘化之后是否能让它运行更大的参数。提示：不能。

解答：[http://thinkpython2.com/code/ackermann\\_memo.py](http://thinkpython2.com/code/ackermann_memo.py)。

### 练习11-4

如果你做过练习10-7，则已经有一个接受了列表作为形参的函数`has_duplicates`，当列表中有任意元素出现多于1次时返回`True`。

使用字典编写一个更快、更简单的`has_duplicates`。

解答：[http://thinkpython2.com/code/has\\_duplicates.py](http://thinkpython2.com/code/has_duplicates.py)。

### 练习11-5

两个单词，如果可以使用轮转操作将一个转换为另一个，则称为“轮转对”（参见练习 8-5中的`rotate_word`函数）。

编写一个程序，读入一个单词表，并找到所有的轮转对。

解答：[http://thinkpython2.com/code/rotate\\_pairs.py](http://thinkpython2.com/code/rotate_pairs.py)。

## 练习11-6

下面是《车迷天下》节目中的另一个谜题  
(<http://www.cartalk.com/content/puzzlers>)：

这个谜题是一个叫Dan O’Leary的伙计寄过来的。他曾经遇到一个单音节、5字母的常用单词，有如下所述的特殊属性。当你删除第一个字母时，剩下的字母组成原单词的一个同音词，即发音完全相同的词。将第一个字母放回去，并删除第二个字母，结果也是原单词另一个同音词。问题是，这个单词是什么？

接下来我给你一个示例，但它并不能完全符合条件。我们看这个5字母单词“**wrack**”，W-R-A-C-K，也就是“**wrack with pain**”（“带来伤害”）里的那个词。如果我删掉第一个字母，会剩下一个4字母的单词，“**R-A-C-K**”。也就是，“**Holy cow, did you see the rack on that buck! It must have a nine-pointer!**”（“天哪！你看到那匹雄鹿的鹿角了吗！一定有9个犄角！”）中的那个词。它是一个完美的同音词。但如果你把“**w**”放回去，并删掉“**r**”，会得到单词“**wack**”，也是一个真实单词，但它读音和其他两个不一样。

但就Dan和我所知，至少有一个单词能够通过删除前两个字母得到两个同音词。问题是，这个单词是什么？

你可以使用练习11-1中的字典来检测一个字符串是否出现在单词表中。

要检查两个单词是不是同音词，可以使用CMU发音词典。你可以从<http://www.speech.cs.cmu.edu/cgi-bin/cmudict> 或者 <http://thinkpython2.com/code/c06d>下载它，也可以下载 <http://thinkpython2.com/code/pronounce.py>，其中提供了一个叫作 `read_dictionary` 的函数来读入发音词典并返回一个Python字典，将每个单词映射到表示其主要读音的字符串上。

编写一个程序，列出所有可以解答这个谜题的单词。

解答：<http://thinkpython2.com/code/homophone.py>。

## 第12章 元组

本章介绍另外一种内置类型——元组，并展示列表、字典和元组三者如何一起工作。我还会介绍一种很有用的可变长参数列表功能：收集操作符和分散操作符。

请注意：元组（tuple）这个词的读音并没有统一标准。有些人会读成“tuh-ple”，与“supple”同音，但在程序设计界，大多数人都读作“too-ple”，与“quadruple”同音。

### 12.1 元组是不可变的

元组是值的一个序列。其中的值可以是任何类型，并且按照整数下标索引，所以从这方面看，元组和列表很像。元组和列表之间的重要区别是，元组是不可变的。

语法上，元组就是用逗号分隔的一系列值：

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

虽然并不必需，但元组常常用括号括起来：

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

若要新建只包含一个元素的元组，需要在后面添加一个逗号：



```
>>> t1 = 'a',  
>>> type(t1)  
<class 'tuple'>
```

而用括号括起来的单独的值并不是元组：

```
>>> t2 = ('a')  
>>> type(t2)  
<class 'str'>
```

新建元组的另一种形式是使用内置函数**tuple**。不带参数时，它会新建一个空元组：

```
>>> t = tuple()  
>>> t  
()
```

如果参数是一个序列（字符串、列表或者元组），结果就是一个包含序列的元素的元组：

```
>>> t = tuple('lupins')  
>>> t  
('l', 'u', 'p', 'i', 'n', 's')
```

因为**tuple** 是内置函数的名称，所以应当避免用它作为变量名称。

大多数列表操作也可以用于元组。方括号操作符可以用下标取得元素：

```
>>> t = ('a', 'b', 'c', 'd', 'e')  
>>> t[0]
```

```
'a'
```

而切片操作符选择一个范围内的元素：

```
>>> t[1:3]  
('b', 'c')
```

但如果尝试修改元组中的一个元素，会得到错误：

```
>>> t[0] = 'A'  
TypeError: object doesn't support item assignment
```

由于元组是不可变的，所以不能修改它的元素。但是可以将一个元组替换为另一个：

```
>>> t = ('A',) + t[1:]  
>>> t  
('A', 'b', 'c', 'd', 'e')
```

这条语句生成新元组，然后使`t` 引用它。

关系运算符适用于元组和其他序列。**Python**从比较每个序列的第一个元素开始。如果它们相等，它就继续比较下一个元素，依次类推，直到它找到不同元素为止。子序列元素不在考虑之列（尽管它们实际上很大）。

```
>>> (0, 1, 2) < (0, 3, 4)  
True  
>>> (0, 1, 2000000) < (0, 3, 4)  
True
```

## 12.2 元组赋值

交换两个变量的值常常很有用。使用传统的赋值方式，需要使用一个临时变量。例如，要交换**a**和**b**：

```
>>> temp = a
>>> a = b
>>> b = temp
```

这种解决方案很笨拙，而**元组赋值**则更优雅：

```
>>> a, b = b, a
```

左边是一个变量的元组，右边是表达式的元组。每个值会被赋值给相应的变量。右边所有的表达式，都会在任何赋值操作进行之前完成求值。

左边变量的个数和右边值的个数必须相同：

```
>>> a, b = 1, 2, 3
ValueError: too many values to unpack
```

更通用地，右边可以是任意类型的序列（字符串、列表或元组）。例如，想要将电子邮件地址拆分成用户名和域名，可以这么写：

```
>>> addr = 'monty@python.org'
>>> uname, domain = addr.split('@')
```

`split` 返回两个元素的列表；第一个元素被赋值到`uname`，第二个到`domain`上。

```
>>> uname
'monty'
>>> domain
'python.org'
```

## 12.3 作为返回值的元组

严格地说，函数只能返回一个值，但如果返回值是元组的话，效果和返回多个值差不多。例如，如果将两个整数相除，得到商和余数，那么先计算`x/y`再计算`x%y`并不高效。更好的方法是同时计算它们。

内置函数`divmod`接收两个参数，并返回两个值的元组，即商和余数。可以将结果存为一个元组：

```
>>> t = divmod(7, 3)
>>> t
(2, 1)
```

或者可以使用元组赋值来分别存储结果中的元素：

```
>>> quot, rem = divmod(7, 3)
>>> quot
2
>>> rem
1
```

下面是返回一个元组的函数的示例：

```
def min_max(t):  
    return min(t), max(t)
```

`max` 和 `min` 都是内置函数，分别返回一个序列的最大值和最小值。`min_max` 计算这两个值并将它们作为一个元组返回。

## 12.4 可变长参数元组

函数可以接收不定个数的参数。以 `*` 开头的参数名会**收集**（gather）所有的参数到一个元组上。例如，`printall` 接收任意个数的参数并打印它们：

```
def printall(*args):  
    print (args)
```

收集参数可以使用任何你想要的名称，但按惯例通常使用 `args`。下面是函数如何工作的一个例子：

```
>>> printall(1, 2.0, '3')  
(1, 2.0, '3')
```

收集的反面是**分散**（scatter）。如果有一个序列的值而想将它们作为可变长参数传入到函数中，可以使用 `*` 操作符。例如，`divmod` 正好接收两个参数，但它不接收元组：

```
>>> t = (7, 3)  
>>> divmod(t)  
TypeError: divmod expected 2 arguments, got 1
```

但如果将元组分散，就可以用了：

```
>>> divmod(*t)
(2, 1)
```

很多内置函数使用可变长参数元组。例如，`max` 和 `min` 都可以接收任意个数的参数：

```
>>> max(1, 2, 3)
3
```

但是 `sum` 并不这样。

```
>>> sum(1, 2, 3)
TypeError: sum expected at most 2 arguments, got 3
```

作为练习，编写一个函数 `sumall`，接收任意个数的参数并返回它们的和。

## 12.5 列表和元组

`zip` 是一个内置函数，接收两个或多个序列，并返回一个元组列表。每个元组包含来自每个序列中的一个元素。这个函数的名字取自拉链（`zipper`），它可以将两行链牙交替连接起来。

下面的例子将字符串和一个列表“拉”到一起：

```
>>> s = 'abc'
>>> t = [0, 1, 2]
>>> zip(s, t)
```

```
<zip object at 0x7f7d0a9e7c48>
```

结果是一个 **zip 对象**，它知道如何遍历每个元素对。使用 **zip** 最常用的方式是在 **for** 循环中：

```
>>> for pair in zip(s, t):  
...     print(pair)  
...  
( 'a', 0)  
( 'b', 1)  
( 'c', 2)
```

**zip** 对象是一种**迭代器**，即用来迭代访问一个序列的对象。迭代器与列表有些方面类似，但与列表不同的是，迭代器不能使用下标来选择对象。

如果需要使用列表的操作符和方法，可以利用 **zip** 对象制作一个列表：

```
>>> list(zip(s, t))  
[( 'a', 0), ( 'b', 1), ( 'c', 2)]
```

结果是一个由元组组成的列表。在本例中，每个元组包含字符串中的一个字符，以及列表中对应的一个元素。

如果序列之间的长度不同，则结果的长度是所有序列中最短的那个：

```
>>> list(zip('Anne', 'Elk'))  
[( 'A', 'E'), ( 'n', 'l'), ( 'n', 'k')]
```

可以在**for** 循环中使用元组赋值来访问元组的列表:

```
t = [('a', 0), ('b', 1), ('c', 2)]
for letter, number in t:
    print (number, letter)
```

每次循环中，Python选择列表中的下一个元组，并将其元素赋值给**letter** 和**number** 变量。这个循环的输出如下:

```
0 a
1 b
2 c
```

如果组合使用**zip** 、**for** 循环以及元组赋值，可以得到一种有用的模式，用于同时遍历两个或更多序列。例如，**has\_match** 函数接收两个序列，**t1** 和**t2**，并当存在一个下标**i** 保证**t1[i] == t2[i]** 时返回**True** :

```
def has_match(t1, t2):
    for x, y in zip(t1, t2):
        if x == y:
            return True
    return False
```

如果需要遍历序列中的元素以及它们的下标，可以使用内置函数**enumerate** :

```
for index, element in enumerate('abc'):
    print(index, element)
```



这个枚举的结果是一个枚举对象，这个对象迭代一个对序列，在这个例子中，每个对都包含一个下标（从0开始）和一个来自给定序列的元素，输出结果还是：

```
0 a
1 b
2 c
```

## 12.6 字典和元组

字典有一个**items** 方法可以返回一个元组的序列，其中每个元组是一个键值对：

```
>>> d = {'a':0, 'b':1, 'c':2}
>>> t = d.items()
>>> t
dict_items([('c', 2), ('a', 0), ('b', 1)])
```

结果是一个**dict\_item** 对象，它是一个迭代器，可以迭代访问每一个键值对。可以使用**for** 循环来访问：

```
>>> for key, value in d.items()
...     print(key, value)
...
c 2
a 0
b 1
```

和预料中一样，字典中的项是没有特定顺序的。

从反方向出发，可以使用一个元组列表来初始化一个新的字典：

```
>>> t = [('a', 0), ('c', 2), ('b', 1)]
>>> d = dict(t)
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

组合使用**dict** 和**zip** 可以得到一个简洁的创建字典的方法：

```
>>> d = dict(zip('abc', range(3)))
>>> d
{'a': 0, 'c': 2, 'b': 1}
```

字典方法**update** 也接收一个元组列表，并将它们作为键值对添加到一个已有的字典中。

使用元组作为字典的键很常见（主要是因为不能使用列表）。例如，一个电话号码簿可能需要将姓名对映射到电话号码。假设定义了**last**，**first** 和**number**，可以这么写：

```
directory[last,first] = number
```

在方括号中的表达式是一个元组。我们也可以使用元组赋值来遍历这个字典：

```
for last, first in directory:
    print(first, last, directory[last,first])
```

这个循环遍历字典**directory** 的所有键，它们都是元组。它将每一个元组的元素赋值给**last** 和**first**，接着打印出名字以及对应的电话号码。

在状态图中有两种方法可以表达元组。更详细的版本和列表一样，显示索引和元素。例如，元组('Cleese', 'John')可以如图12-1所示。

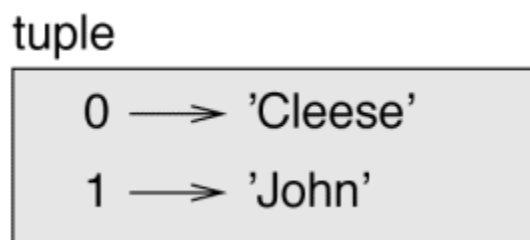


图12-1 状态图

但是在更大的图中你可能希望省略掉细节。例如，整个电话簿的图如图12-2所示。

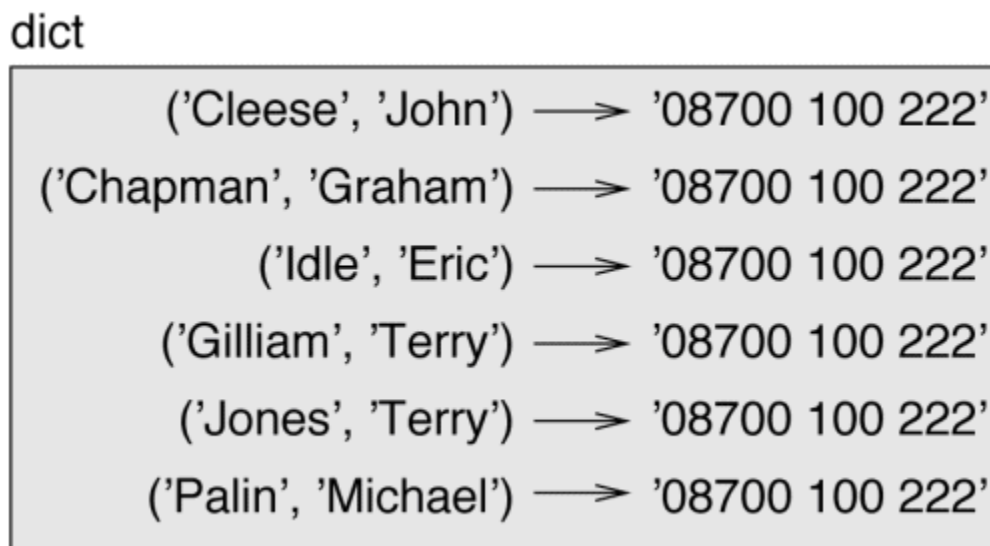


图12-2 状态图

这里元组使用Python的语法作为图形化的简写展示。这张图里的电话号码是BBC的投诉热线，所以请不要真去拨打它。

## 12.7 序列的序列

我一直在聚焦于元组的列表，但本章中几乎所有的示例都可以对列表的列表、元组的元组以及列表的元组使用。为了避免枚举所有的可能组合，有时候直接说序列的序列更简单。

在很多环境中，不同类型的序列（字符串、列表和元组）都可以互换使用。应当如何选择使用哪个呢？

从最明显的一个开始，字符串比其他序列有更多限制，因为它的元素必须是字符。它们也是不可变的。如果你需要修改一个字符串中的字符（而不是新建一个字符串），可能需要使用字符的列表。

列表比元组更加通用，主要因为它是可变的。但也有一些情况下你可能会优先选择元组。

1. 在有些环境中，如返回语句中，创建元组比创建列表从语法上说更容易。
2. 如果需要用序列作为字典的键，则必须使用不可变类型，如元组或字符串。
3. 如果你要向函数传入一个序列作为参数，使用元组可能会减少潜在的由假名导致的不可预知行为。

因为元组是不可变的，它们不提供类似`sort`和`reverse`之类的方法，这些方法修改现有的序列。但Python也提供了内置函数`sorted`，可以接收任何序列作为参数，并按排好的顺序返回带有同样元素的

新列表。Python还提供了**reverse**，可以接收序列作为参数，并返回一个以相反顺序遍历列表的迭代器。

## 12.8 调试

列表、字典和元组都被统一看作是一种**数据结构**。本章中我们开始看到复合数据结构，像元组的列表，或者用元组做键、用列表做值的字典等。复合数据结构很有用，但它容易导致我称为的**结构错误**；也就是说，数据结构因为错的类型、大小或结构导致的错误。例如，如果你期望得到一个包含单个整数的列表，而我给你一个单个整数（而不是在列表中），就会出错。

为了帮助调试这种问题，我写了一个模块**structshape**，提供一个也叫作**structshape**的函数，接收任何数据类型作为参数，并返回一个描述它的形状的字符串。你可以从<http://thinkpython2.com/code/structshape.py>下载它。

下面是一个简单列表的结果：

```
>>> from structshape import structshape
>>> t = [1,2,3]
>>> structshape(t)
'list of 3 int'
```

更好看的程序可能会输出“list of 3 ints”，但不需要处理复数更加容易。下面是列表的列表：

```
>>> t2 = [[1,2], [3,4], [5,6]]
>>> structshape(t2)
```

```
'list of 3 list of 2 int'
```

如果列表的元素不是同一种类型，`structshape` 会根据它们的类型按顺序分组：

```
>>> t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
>>> structshape(t3)
'list of (3 int, float, 2 str, 2 list of int, int)'
```

下面是元组的列表：

```
>>> s = 'abc'
>>> lt = list(zip(t, s))
>>> structshape(lt)
'list of 3 tuple of (int, str)'
```

下面是一个字典，有3个从整数映射到字符串的项：

```
>>> d = dict(lt)
>>> structshape(d)
'dict of 3 int->str'
```

如果你发现要记住数据结构有困难，`structshape` 可以帮忙。

## 12.9 术语表

元组（**tuple**）：一个不可变的元素序列。

元组赋值（**tuple assignment**）：一个赋值语句，右侧是一个序列，左侧是一个变量的元组。右边的序列会被求值，它的元素依次赋值给左侧元组中的变量。

收集 (**gather**) : 组装可变长参数元组的操作。

分散 (**scatter**) : 把一个序列当作参数列表的操作。

**zip** 对象 (**zip object**) : 调用内置函数**zip** 的结果, 它是一个迭代访问由元组组成的序列的对象。

迭代器 (**iterator**) : 可以遍历序列的对象, 但它不提供列表的操作和方法。

数据结构 (**data structure**) : 相关的值的集合, 通常组织成列表、字典、元组等。

结构错误 (**shape error**) : 某个值由于其结构不对导致的错误, 即它的类型或尺寸不对。

## 12.10 练习

### 练习12-1

编写一个函数**most\_frequent**, 接收一个字符串并按照频率的降序打印字母。从不同语言中查找文本样例并查看不同语言中的单词频率如何变化。将你的结果和

[http://en.wikipedia.org/wiki/Letter\\_frequencies](http://en.wikipedia.org/wiki/Letter_frequencies)上的列表进行对比。

解答: [http://thinkpython2.com/code/most\\_frequent.py](http://thinkpython2.com/code/most_frequent.py)。

### 练习12-2

更多回文！

1. 编写一个程序从文件中读入一个单词列表（参见9.1节）并打印出所有是回文的单词集合。

下面是输出的样子的示例：

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']  
['retainers', 'ternaries']  
['generating', 'greatening']  
['resmelts', 'smelters', 'termless']
```

提示：你可能需要构建一个字典将字母的集合映射到可以用这些字母构成的单词的列表上。问题是，如何表达字母集合，才能让它可以用作字典的键？

2. 修改前一个问题的程序，让它先打印最大的回文列表，再打印第二大的回文列表，依次类推。

3. 在Scrabble拼字游戏中，一个“bingo”代表你自己架子上全部7个字母和盘上的一个字母组合成一个8字母单词。哪一个8字母单词可以生成最多的bingo？提示：一共有7个。

解答：[http://thinkpython2.com/code/anagram\\_sets.py](http://thinkpython2.com/code/anagram_sets.py)。

### 练习12-3

两个单词，如果可以通过交换两个字母将一个单词转换为另一个，就称为“置换对”；例如，“converse”和“conserve”。编写一个程序



查找字典中所有的置换对。提示：不要测试所有的单词对，也不要测试所有可能的交换。

解答：<http://thinkpython2.com/code/metathesis.py>。

鸣谢：这个练习启发自<http://puzzlers.org>的示例。

## 练习12-4

下面是《车迷天下》节目中的一个谜题  
(<http://www.cartalk.com/content/puzzlers>)：

一个英文单词，当逐个删除它的字母时，仍然是英文单词。这样的单词中最长的是什么？

首先，字母可以从两头或者中间删除，但你不能重排字母。每次你去掉一个字母，则得到另一个英文单词。如果一直这么做，最终会得到一个字母，它本身也是一个英文单词——可以从字典上找到的。我想知道这样的最长的单词是什么，它有多少字母？

我会给你一个普通的例子：Sprite。你从sprite开始，取出一个字母，从单词内部取，取走r，这样我们就剩下单词spite，接着我们取走结尾的e，剩下spit，接着取走s，我们剩下pit、it和I。

编写一个程序来找到所有可以这样缩减的单词，然后找到最长的一个。

这个练习比大部分练习都更有挑战，所以下面有一些建议。

1. 你可能需要编写一个程序接收一个单词，并计算出所有通过从它取出一个字母得到的单词的列表。它们是这个单词的“子”单词。

2. 递归地，只有当一个单词的子单词中有一个可缩减时，它才可缩减。作为一个基准情形，你可以认为空字符串可缩减。

3. 我提供的单词表，`words.txt`，并不存在单个字母的单词。所以你可能需要加上“`I`”、“`a`”和空字符串。

4. 为了提高程序的效率，你可能需要记住已知的可缩减的单词。

解答：<http://thinkpython2.com/code/reducible.py>。

## 第13章 案例研究：选择数据结构

到这里你应该已经学会了Python的核心数据结构，也见过了一些使用它们的算法。如果你想要更多地了解算法，可以阅读第21章。但继续下面的内容之前那部分内容并不是必需要读懂，你可以随感兴趣时时去阅读。

本章配合练习介绍一个案例分析，帮你思考如何选择数据结构并如何实际使用它们。

### 13.1 单词频率分析

和前面的章节一样，应当至少尝试一下解决问题，再看我的解答。

#### 练习13-1

编写一个程序，读入一个文件，将每行内容拆解为单词，剥去单词周围的空白字符和标点，并转换为小写。

提示：`string` 模块提供了空白字符串`whitespace`，包括空格、制表符、换行符等；它也提供了`punctuation`，包含了所有的标点字符。让我们试试能不能让Python胡言乱语：

```
>>> import string
>>> string.punctuation
```

```
'! "$%&'()*+,-./:;<=>?@[\\]^_`{|}~'
```

另外，也可以考虑字符串方法`strip`、`replace`和`translate`。

## 练习13-2

去往古腾堡工程（Project Gutenberg, <http://www.gutenberg.org>）并下载你最喜欢的无版权书籍的纯文本文档。

修改前一个练习中的程序，改为从你下载的书籍中读取内容，跳过文件开头的信息部分，并和前面一样将文本处理成为单词。

接着修改程序，计算书中出现的全部单词的总数，以及每个单词使用的次数。

打印书中使用的不同单词的个数。比较不同时代、不同作者的不同书籍。哪一个作者使用的词汇最广泛？

## 练习13-3

修改前一个练习中的程序，计算书中使用频率最高的20个单词。

## 练习13-4

修改前面的程序，读入一个单词表（参见9.1节）并打印出书中所有不在单词表之中的单词。这其中有多少是拼写错误？有多少是应该出现在单词表中的常用单词？有多少是真正冷僻的单词？

## 13.2 随机数

给定相同的输入，大部分计算机程序每次运行都会生成相同的输出，所以它们被认为是有**确定性**的。确定性通常是件好事，因为我们希望相同的计算能有相同的结果。但对某些特别的应用，我们希望计算机是不可预测的。游戏是一个明显的例子，但还有更多类似的例子。

让程序变得真正地不确定很难，但也有办法让它至少看起来是不确定的。一种办法是使用算法来生成**伪随机数**。伪随机数并不是真正随机的，因为它们是通过一个确定性的算法生成的，但若只看输出的数字的话，几乎不可能看出来和随机数有什么区别。

模块**random** 提供了用于生成伪随机数的函数（接下来我直接简单地将它称为“随机数”）。

函数**random** 返回一个从0.0到1.0之间的随机浮点数（包括0.0，但不包括1.0）。每当调用**random** 时，会得到一个很长的随机数序列中的下一个数。运行下面的循环，可以看到一个样本：

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

函数**randint** 接收参数**low** 和**high**，并返回**low** 和**high** 之间（两者都包含）的一个整数。

```
>>> random.randint(5, 10)
5
>>> random.randint(5, 10)
9
```

要从序列中随机选择一个元素，可以使用`choice`：

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
3
```

`random` 模块还提供了可以从各种连续分布序列中生成随机数的函数，包括高斯分布、指数分布、 $\gamma$  分布以及其他几种。

### 练习13-5

编写一个函数`choose_from_hist`，接收一个11.1节所定义的直方图作为参数，并从这个直方图中，按照频率的大小，成比例地随机返回一个值。例如，对下面这个直方图：

```
>>> t = ['a', 'a', 'b']
>>> hist = histogram(t)
>>> hist
{'a': 2, 'b': 1}
```

你的函数应该以2/3的概率返回'`a`'，以1/3的概率返回'`b`'。

## 13.3 单词直方图

在继续阅读之前你应当尝试前面的练习。你可以从 [http://thinkpython2.com/code/analyze\\_book1.py](http://thinkpython2.com/code/analyze_book1.py) 下载我的解答。你还需要 <http://thinkpython2.com/code/emma.txt>。

下面是一个读取文件并从文件中的单词构造直方图的例子：

```
import string

def process_file(filename):
    hist = dict()
    fp = open(filename)
    for line in fp:
        process_line(line, hist)
    return hist

def process_line(line, hist):
    line = line.replace('-', ' ')

    for word in line.split():
        word = word.strip(string.punctuation + string.whitespace)
        word = word.lower()

        hist[word] = hist.get(word, 0) + 1

hist = process_file('emma.txt')
```

这个程序读入 `emma.txt`，其内容是简·奥斯丁的《爱玛》的文本。

`process_file` 循环遍历文件中的每一行，每次将一行传递给 `process_line` 函数。直方图 `hist` 用作累加器。

`process_line` 使用字符串方法 `replace` 将 `' - '` 符号替换为空格，再使用 `split` 将各行文本拆分成一个字符串列表。它遍历单词列表，使用 `strip` 和 `lower` 去除掉标点符号并转换为小写。（我们

说“转换”，只是个简称，记住字符串是不可变的，所以`strip`和`lower`这样的方法返回的是新字符串。）

最后，`process_line`通过创建新项或者增加旧有项的值来更新直方图。

要计算文件中单词的总数，我们可以累加直方图中的频率：

```
def total_words(hist):  
    return sum(hist.values())
```

不同单词的个数，就是字典里的元素数量：

```
def different_words(hist):  
    return len(hist)
```

下面是打印结果的代码：

```
print('Total number of words:', total_words(hist))  
print('Number of different words:', different_words(hist))
```

以及结果：

```
Total number of words: 161080  
Number of different words: 7214
```

## 13.4 最常用的单词



要寻找最常用单词，我们可以生成一个元组的列表，其中每个元组包括一个单词及其频率，并对其进行排序。

下面的函数接收一个直方图，并返回“单词-频率”元组的列表：

```
def most_common(hist):
    t = []
    for key, value in hist.items():
        t.append((value, key))

    t.sort(reverse=True)
    return t
```

在每个元组中，频率先出现，所以结果列表按频率排序。下面的循环打印出最常用的10个单词：

```
t = most_common(hist)
print('The most common words are:')
for freq, word in t[:10]:
    print(word, freq, sep='\t')
```

这里我使用关键字参数`sep` 通知`print` 去使用制表符作为分隔符，而不使用空格。于是第二列可以对其排列。下面是《爱玛》的结果：

```
The most common words are:
to      5242
the     5205
and     4897
of      4295
i       3191
a       3130
it      2529
her     2483
was     2400
she     2364
```

这段代码可以用`sort` 函数的`key` 参数进行简化。如果你有兴趣，可以读一下相关的文章：

<http://wiki.python.org/moin/HowTo/Sorting>。

## 13.5 可选形参

我们已经见过一些接收可选形参的内置函数和方法。用户也可以编写接收可选形参的自定义函数。例如，下面的函数打印一个直方图中最常见的单词：

```
def print_most_common(hist, num=10):  
    t = most_common(hist)  
    print('The most common words are:')  
    for freq, word in t[:num]:  
        print(word, freq, sep='\t')
```

第一个形参是必需的；第二个是可选的。形参`num`的**默认值**是10。

如果只提供一个实参：

```
print_most_common(hist)
```

`num` 会获得默认值。如果提供两个实参：

```
print_most_common(hist, 20)
```

`num` 则会获得所提供的实参值。换句话说，可选实参值**覆盖**默认形参值。

如果一个函数既有必需形参，也有可选形参，则所有的必需形参都必须在前面，后面跟着可选形参。

## 13.6 字典减法

寻找在书中出现却不在`words.txt` 单词表中的单词，这个问题可以看作是集合减法；也就是说，我们想要找到出现在一个集合（书中的单词）而不在另一个集合（单词表中的单词）的所有单词。

`subtract` 函数接收两个字典`d1` 和`d2`，并返回一个新的字典，包含所有出现在`d1` 中且不出现在`d2` 中的键值。由于我们并不真的关心字典的值，我们将所有值都设为`None`。

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

要找出书中出现而不在`words.txt` 单词表中的词，我们可以使用`process_file` 为`words.txt` 建立一个直方图，再使用减法：

```
words = process_file('words.txt')
diff = subtract(hist, words)

print("Words in the book that aren't in the word list:")
for word in diff:
    print(word, end=' ')
```

下面是《爱玛》一书中的部分结果：

```
Words in the book that aren't in the word list:
rencontre jane's blanche woodhouses disingenuousness
friend's venice apartment ...
```

这些词中有些是名字或所有格单词。其他的，如“rencontre”，已经不再常用。但也有一些是真应该包含在单词表中的！

### 练习13-6

Python提供了一个数据结构**set**，它提供了很多常见的集合操作。你可以读19.5节中关于集合操作的内容，或者阅读<http://docs.python.org/3/library/stdtypes.html#types-set>上的文档，并编写一个程序使用集合减法来寻找出现在书中但不出现在单词表中的单词。解答：[http://thinkpython2.com/code/analyze\\_book2.py](http://thinkpython2.com/code/analyze_book2.py)。

## 13.7 随机单词

若要从直方图中随机选择一个单词，最简单的算法是根据计算得到的频率构建一个列表，其中每个单词根据词频有多个拷贝，并从中随机选择一个单词：

```
def random_word(h):
    t = []
    for word, freq in h.items():
        t.extend([word] * freq)

    return random.choice(t)
```

表达式`[word] * freq`创建一个列表，里面有单词`word`的`freq`个副本。`extend`方法和`append`类似，区别是接收的参数是一

个序列。

这个算法可以使用，但效率并不高；每当选择一个随机单词时，它会重建列表，而这个列表和原书差不多长。一个明显的改进方法是只建立列表一次，再使用多次选择，但这么做列表仍然很大。

更好的替代方案如下。

1. 使用`keys`来获得书中所有的单词的列表。
2. 构建一个列表，包含单词频率的累积和（参见练习10-2）。这个列表中的最后一项是书中单词的总数 $n$ 。
3. 在1到 $n$ 之间随机选择一个数。使用二分查找（参见练习10-11）来找到随机数在累积和列表中应该出现的位置的下标。
4. 使用这个下标，在单词表中找到相应的单词。

### 练习13-7

编写一个程序，使用这个算法来从书中选择一个随机的单词。

解答：[http://thinkpython2.com/code/analyze\\_book3.py](http://thinkpython2.com/code/analyze_book3.py)。

## 13.8 马尔可夫分析

如果你从书中随机地获取单词，可以借此感受一下书中的词汇，但可能无法通过随机获取来得到一句话：

this the small regard harriet which knightley's it most things

一个随机单词的序列，很难组成有意义的话，因为相邻的词之间没有任何关联。例如，在一个真实的句子中，冠词“the”应当会后接一个形容词或名词，而不应是动词或副词。

测量这种类型的关联的方法之一是使用马尔可夫分析，它能够用于描述给定的单词序列中下一个可能出现的单词的概率。例如，歌曲《Eric, the Half a Bee》的开头是：

Half a bee, philosophically,

Must, ipso facto, half not be.

But half the bee has got to be

Vis a vis, its entity. D’you see?

But can a bee be said to be

Or not to be an entire bee

When half the bee is not a bee

Due to some ancient injury?

在这段文本中，短语“half the”总是后接着单词“bee”，但短语“the bee”则可能后接“has”或“is”。

马尔可夫分析的结果是一个从每个前缀（如“half the”和“the bee”）到其所有可能后缀（如“has”和“is”）的映射。

给定这种映射后，你就可以用它来生成随机文本。从任意前缀开始，并从它的可能后缀中随机选择一个。接着，你可以将前缀的结尾和后缀组合起来，作为下一个前缀，并继续重复。

例如，如果你以前缀“Half a”开始，则接下来一个单词必定是“bee”，因为这个前缀在文本中只出现了一次。下一个前缀是“a bee”，所以下一个后缀可能是“philosophically”“be”或者“due”。

在这个例子中前缀的长度总是2，但其实你可以使用任意前缀长度来进行马尔可夫分析。

### 练习13-8

马尔可夫分析：

1. 编写一个程序从文件中读入文本，并进行马尔可夫分析。结果应该是一个字典，将前缀映射到可能后缀的集合。集合可以是列表、元组或者字典；由你来做出合适的选择。你可以使用前缀长度2来测试程序，但编写程序时应当考虑可以方便地改为其他前缀长度。

2. 在前面编写的程序中添加一个函数，基于马尔可夫分析的结果随机生成文本。下面是一个从《爱玛》中使用前缀长度2生成的例子：

He was very clever, be it sweetness or be angry, ashamed or only amused, at such a stroke. She had never thought of Hannah till you were

never meant for me?” “I cannot make speeches, Emma.” he soon cut it all himself.

对这个例子，我留下了每个单词后面的标点。结果几乎是语法正确的，但也不完全对。语义上，它看起来很像是有意义的，但也不完全是。

当增加前缀长度时，结果会怎么样？随机生成的文本会不会看来更有意义？

3. 一旦你的程序可以正常运行后，可以考虑尝试一下混搭：如果对两本或更多本书进行组合，则生成的随机文本会以一种有趣的方式混合各书中的词汇和短语。

致谢：本案例分析基于Kernighan和Pike的*The Practice of Programming*（Addison-Wesley, 1999）一书中的一个示例。

你应当在继续阅读前尝试这个练习，接着可从  
<http://thinkpython2.com/code/markov.py>下载我的解答。你也需要  
<http://thinkpython2.com/code/emma.txt>。

## 13.9 数据结构

使用马尔可夫分析生成随机文本很有趣，但这个练习还有一个要点：数据结构的选择。在前面的练习中，你需要选择：

- 如何表达前缀；
- 如何表达可能的后缀的集合；



- 如何表达每个前缀到可能后缀的集合的映射。

最后一个选择很简单，要从键映射到对应的值，字典是最自然的选择。

对前缀来说，最明显的选择是字符串、字符串列表或者字符串元组。对后缀来说，一种选择是列表，另一种是直方图（字典）。

你会如何选择？第一步需要思考每种数据结构需要实现的操作。对前缀而言，我们需要能够从前方删除一个单词，并在后方添加一个单词。例如，如果当前的前缀是“Half a”，而下一个单词是“bee”，则需要能够构造下一个前缀，“a bee”。

你的第一个选择可能是列表，因为列表添加和删除元素都很方便。但我们也需要使用前缀作为字典的键，所以列表被排除掉。对元组而言，虽然你不能附加或删除，但可以使用加法操作符来构建一个新的元组：

```
def shift(prefix, word):  
    return prefix[1:] + (word,)
```

`shift` 接收一个单词的元组、`prefix`，以及一个字符串`word`，并构建一个新的元组，包含`prefix`中除了第一个之外的元素，并把`word`添加在最后。

对后缀集合而言，我们需要进行的操作包括添加一个新的后缀（或者增加一个已有后缀的频率），以及随机选择一个后缀。

添加一个新后缀，使用列表实现或者直方图实现效率上相同。从一个列表中随机选择元素很简单；从直方图中随机选择则更难一些（参见练习13-7）。

到此为止我们一直在讨论实现的简易性，但选择数据结构时，还有其他需要考虑的因素。一个是运行时间。有时候，我们可以从理论上预期一种数据结构比另一种更快；例如，我提到过`in`操作符，当元素数量很大时，在字典中使用比在列表中快。

但哪种实现会更快常常无法事先预知。一个办法是两种都实现，再比较哪个更快。这种方法称为**基准比较**（**benchmarking**）。比较实际的方案是先选择最容易实现的数据结构，然后看它是否对预期的程序而言足够快。如果已经足够，则不需要变动；否则，可以使用 **profile** 模块之类的工具，发现程序中哪些地方占用了最长的时间。

另一个考虑因素是存储空间。例如，使用直方图来保存后缀集合可能占用较少空间，因为不论一个单词在文本中出现多少次，你只需要保存一次。有的情况下，节省空间也可以让你的程序运行更快，而在极端的情形中，如果导致内存溢出，则程序无法正常运行。但对大多数程序来说，存储空间是次于运行速度的第二考虑因素。

最后一点：在这个讨论中，对于分析和生成两个过程，我暗示了我们应当使用相同的数据结构。但因为这是两个分开的阶段，所以也可以在分析阶段使用一种数据结构，再转换为另一种数据结构用于生成阶段。如果新的数据结构在生成阶段节省的时间大于转换花费的时间，则总的来说是有利的。

## 13.10 调试

当在调试程序时，尤其是对付一个困难的bug时，可以尝试下面5点。

### 阅读

审阅你的代码，对自己读出来，并检查它是否和你想说的一致。

### 运行

做一些小修改并进行试验，或者运行不同的版本。通常如果在程序中正确的地方加上正确的输出，问题就会变得更加显而易见。但有时候你需要构建一个脚手架。

### 沉思

花些时间思考！可能是哪种类型的错误：语法的、运行时的还是语义的？从错误消息或程序输出中可以得到什么信息？哪种错误可能导致你看到的问题？在问题出现之前，你的最后一次修改是什么？

### 橡皮鸭调试

如果你向其他人解释遇到的问题，有时候能在说完问题之前就找到答案。通常你甚至不需要去找人去诉说，而只需要对橡皮鸭诉说即

可。这就是著名的橡皮鸭调试（rubber duck debugging）的来源。这可不是我编出来的，参见[https://en.wikipedia.org/wiki/Rubber\\_duck\\_debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging)。

## 回退

在某种情况下，最好的办法就是回退，撤销最近的修改，直到你的程序恢复到之前没有错误且能够理解的程度。然后可以开始重新构建。

新手程序员有时会卡在这些环节中的某一个上，却忘了还可以尝试其他的环节。每个环节都有其独自的失败模式。

例如，当问题是一个拼写错误时，阅读代码可以帮忙，但若问题是概念误解导致，就没有效果了。如果你不理解自己的程序，那么即使阅读100遍，也发现不了问题，因为错误是在你大脑中的。

运行一些试验代码可以起到很大帮助，尤其是那些短小而简单的测试程序。但如果你没有思考或阅读代码就运行试验代码，则可能会陷入我称之为“随机走动编程”的模式之中。即毫无目标地随机改变程序，直到程序正确运行为止。毫无疑问，随机走动编程可能要花费很长的时间。

你必需花一定的时间去思考。调试就像是一门实验科学。你应当至少有一个关于这个问题的假设。如果有两个以上的可能性，可以试着构思一个测试来排除其中一个。

但如果有很多错误，或者你要修正的代码太大太复杂，即使最好的调试技巧也会失败。有时候最好的选择是回退，简化程序，直到得到一个你能够理解并且正确运行的程序。

新手程序员往往不愿意后撤，他们无法忍受删除一行代码（即使那是错误的代码）。如果能让你感觉更好，可以将程序复制到另外一个文件再开始删减它。这样以后就可以一点一点地复制回来。

寻找一个困难的bug，需要阅读、运行、沉思，甚至有时候需要回退。如果你在这其中一个环节上卡住了，可以尝试其他的环节。

## 13.11 术语表

**确定性 (deterministic)**：程序的一种特性：给定相同的输入，每次运行都会执行相同的操作。

**伪随机 (pseudorandom)**：一序列数：看起来是随机的，但实际上是由带着确定性的程序生成的。

**默认值 (default value)**：可选形参声明时给定的值，如果函数调用时没有指定这个实参的值，则使用该默认值。

**覆盖 (override)**：使用实参值替换一个默认值。

**基准测试 (benchmarking)**：实现不同的备选方案，并使用各种可能输入的样本来测试它们，以达到选择使用哪种数据结构的目的。

橡皮鸭调试（rubber duck debugging）：通过向类似橡皮鸭之类的静物解释你的问题，进行调试的过程。虽然橡皮鸭不懂Python，但通过诉说和解释，可以帮助你解决问题。

## 13.12 练习

### 练习13-9

一个单词的“排名”是它在单词列表中按频率排序的位置：最常见的词排名第1，次常用的词排第2，等等。

齐普夫定律（Zipf's law）描述了排名和自然语言中词频的关系（[http://en.wikipedia.org/wiki/Zipfs\\_law](http://en.wikipedia.org/wiki/Zipfs_law)）。特别地，它预测了排名为 $r$ 的单词的频率 $f$ ：

$$f = cr^{-s}$$

这里 $s$ 和 $c$ 是依赖于语言和文本的参数。如果在表达式两侧都调用对数，则得到：

$$\log f = \log c - s \log r$$

所以如果以 $\log r$ 为横轴给 $\log f$ 绘图，则会得到斜率为 $-s$ ，截距为 $\log c$ 的直线。

编写一个程序，从文件中读入文本，计算单词词频，并按照词频的降序，每一行打印出一个单词，以及 $\log f$ 和 $\log r$ 。使用你喜欢的制

图程序将结果以图表形式展现出来，并检查它是否为直线。你能估计s的值吗？

解答：<http://thinkpython2.com/code/zipf.py>。要运行我的解答，你需要安装绘图模块`matplotlib`。如果安装过Anaconda，你就已经有了`matplotlib`，否则你可能需要安装它。

## 第14章 文件

本章介绍“持久”程序的概念，它们将数据存储到持久存储中。另外，我们还会看到不同种类的持久存储，如文件和数据库。

### 14.1 持久化

我们现在见过的程序都是瞬态的，因为它们会在短暂的时间里运行出一些输出，但当运行结束后，它们的数据会消失。如果再次运行程序，它会再次全新地开始。

也有些程序是**持久化**的：它们会运行很长一段时间（或者一直运行）；它们会至少存储一部分数据到永久存储（例如，硬盘）中；而且如果它们被关闭重启后，会接着从上次离开的状态继续。

持久化程序的例子包括操作系统，它几乎运行在任何一台开启的电脑中，以及web服务器，它们通常持续运行，等待网络上连入的请求。

读写文本文件是程序维护数据最简单的方法之一。我们已经见过读取文本文件的程序；在本章中将会见到往文件写入的程序。

另一种办法是将程序的状态保存到数据库中。本章中我们会介绍一个简单的数据库，以及一个模块，`pickle`，用来简化程序数据的存储。



## 14.2 读和写

文本文件是存储在诸如硬盘、闪存或光盘的永久媒介上的字符串序列。我们已经在9.1节中见过如何打开和读取一个文件。

要写入一个文件，需要使用 **'w'** 模式作为第二个实参来打开它：

```
>>> fout = open('output.txt', 'w')
```

如果文件已经存在，则使用写模式打开时会清除掉旧有数据并重新开始，所以请谨慎！如果文件不存在，则会新建一个。

**open** 函数返回一个文件对象，提供操作文件的方法。其中 **write** 方法把数据写入到文件中。

```
>>> line1 = "This here's the wattle,\n"
>>> fout.write(line1)
24
```

返回值是写入的字符数目。文件对象会记录写到了哪里，所以如果你再次调用 **write**，它会在文件的结尾处添加新的数据。

```
>>> line2 = "the emblem of our land.\n"
>>> fout.write(line2)
24
```

当写入完毕时，应该关闭文件。

```
>>> fout.close()
```

如果不关闭文件，程序会在执行结束时将文件关闭。

## 14.3 格式操作符

`write` 的参数必须是字符串，所以若我们想要往文件中写入其他类型的值，必须将它们先转换为字符串。最容易的办法是使用 `str`：

```
>>> x = 52
>>> fout.write(str(x))
```

另一个办法是使用**格式操作符**`%`。当用于整数时，`%` 是求余操作符。但若第一个操作对象是字符串时，`%` 则是格式操作符。

`%` 的第一个操作对象是**格式字符串**，包括了一个或多个**格式序列**，由它们来指定第二个操作对象如何格式化。表达式的结果是一个字符串。

例如，格式序列 `'%d'` 意味着第二个操作数应该被格式化为十进制整数。

```
>>> camels = 42
>>> '%d' % camels
'42'
```

结果是字符串 `'42'`，请不要将它和整数值42混淆。

格式序列可以出现在字符串的任意地方，所以可以在一个句子中嵌入变量值：

```
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

如果字符串中有多于一个格式序列，第二个操作对象就必须是元组。每个格式序列按顺序对应元组中的一个元素。

下面的例子使用 '%d' 格式化整数， '%g' 格式化浮点数，以及 '%s' 格式化字符串：

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

元组中元素的个数必须和字符串中格式序列的个数一致。另外，元素的类型也要和格式序列一致：

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: %d format: a number is required, not str
```

第一个例子中，元组中元素个数不够；第二个例子中，元素的类型不对。

更多关于格式操作符的信息参见 <https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting>。还有一个更强大的替代方案是字符串格式方法，参见 <https://docs.python.org/3/library/stdtypes.html#str.format>。

## 14.4 文件名和路径

文件组织在**目录**（也称为文件夹）中。每个程序都有“当前目录”，它是大多数操作的默认目录。例如，当打开一个文件用于读取时，Python默认在当前目录寻找它。

os 模块提供了用于操作文件和目录的函数（os代表operating system，即操作系统）。os.getcwd 返回当前目录的名称：

```
>>> import os
>>> cwd = os.getcwd()
>>> cwd
'/home/dinsdale'
```

cwd 表示current working directory（即“当前工作目录”）。这个例子里的结果是/home/dinsdale，是名为dinsdale的用户的主目录。

类似于'/home/dinsdale' 这样用来定位一个文件或目录的字符串被称为一个**路径**（path）。

而一个简单文件名，如memo.txt，也被认为是一个路径，但它是一个相对路径，因为它依赖于当前目录。如果当前目录是/home/dinsdale，则文件名memo.txt指的是/home/dinsdale/memo.txt。

而以/开头的路径则不依赖于当前目录，所以被称为**绝对路径**（absolute path）。可以使用os.path.abspath 来找寻文件的绝对路径：

```
>>> os.path.abspath('memo.txt')
'/home/dinsdale/memo.txt'
```

`os.path` 还提供了其他函数来操作文件名和路径。例如，`os.path.exists` 检查一个文件或目录是否存在：

```
>>> os.path.exists('memo.txt')
True
```

如果它存在，`os.path.isdir` 可以检查它是否为目录：

```
>>> os.path.isdir('memo.txt')
False
>>> os.path.isdir('/home/dinsdale')
True
```

类似地，`os.path.isfile` 检查它是否为文件。

`os.listdir` 返回指定目录中文件（以及其他目录）的列表：

```
>>> os.listdir(cwd)
['music', 'photos', 'memo.txt']
```

为了演示这些函数，下面的例子“走遍”一个目录，打印所有文件的名称，并对之中的子目录递归调用自己。

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        else:
```

```
walk(path)
```

`os.path.join` 接收一个目录和一个文件名称，并将它们拼接为一个完整的路径。

`os` 模块提供了一个函数`walk`，和上面的例子作用类似，但功能更丰富。作为练习，请阅读文档，并使用它打印指定目录中文件的名称和它的子目录。你可以从<http://thinkpython2.com/code/walk.py>下载我的解答。

## 14.5 捕获异常

当尝试读取和写入文件时，很多东西都可能出错。如果尝试打开一个不存在的文件，会得到一个`IOError`：

```
>>> fin = open('bad_file')
IOError: [Errno 2] No such file or directory: 'bad_file'
```

如果没有权限访问一个文件：

```
>>> fout = open('/etc/passwd', 'w')
PermissionError: [Errno 13] Permission denied: '/etc/passwd'
```

如果尝试打开一个目录用于文件读取，会得到：

```
>>> fin = open('/home')
IsADirectoryError: [Errno 21] Is a directory: '/home'
```

要避免这些错误，可以使用类似`os.path.exists` 和 `os.path.isfile` 的函数，但要检查所有的可能需要花费大量时间和代码（“Errno 21”这个名字，说明至少有21种可能出错的地方）。

最好是直接去尝试——等发生问题时再去解决它们——这也正是 `try` 语句所做的事情。语法和 `if...else` 语句类似：

```
try:
    fin = open('bad_file')
except:
    print ('Something went wrong.')
```

Python会先从`try` 子句开始，如果一切顺利，则跳过`except` 语句并继续执行。如果发生了异常，则跳出`try` 子句，并运行`except` 子句。

使用`try` 语句处理异常的过程称为**捕获**一个异常。在这个例子里，`except` 语句打印的错误信息并没有太多用处。总的来说，捕获异常给了你一个修补错误的机会，或者可以再次尝试，或者至少能够优雅地停止程序。

## 14.6 数据库

**数据库** 是一个有组织的用于存储数据的文件。许多数据库都像字典一样组织数据，因为它们也将键映射到值上。数据库和字典之间最大的区别是数据库是保存在磁盘上（或者其他永久存储上）的，所以当程序结束时它也能持续存在。

模块**dbm** 提供了接口用于创建和更新数据库文件。作为示例，我将会创建一个数据库保存图片文件的标题。

打开一个数据库和打开其他类型的文件差不多：

```
>>> import dbm
>>> db = dbm.open('captions', 'c')
```

模式 '**c**' 意味着数据库应当被创建，如果它不存在的话。调用的结果是一个数据库对象，（对大多数操作）可以当作字典来用。

当创建一个新项时，**dbm** 会更新数据库文件。

```
>>> db['cleese.png'] = 'Photo of John Cleese.'
```

当访问数据库中的一项时，**dbm** 会读取文件：

```
>>> db['cleese.png']
b'Photo of John Cleese.'
```

这里的结果是一个**字节组对象**（bytes object），因此以**b** 开头。字节组对象和字符串很类似。当你更加深入研究Python的时候，它们的区别可能会变得很重要，但现在可以忽略。

如果对一个已经存在的键赋值，**dbm** 会替换旧值：

```
>>> db['cleese.png'] = 'Photo of John Cleese doing a silly walk.'
>>> db['cleese.png']
b'Photo of John Cleese doing a silly walk.'
```



有一些字典方法，如**keys** 和**items**，对数据库对象不可以使用。但使用**for** 循环来迭代遍历是可以的：

```
for key in db:
    print(key, db[key])
```

和其他文件一样，当操作结束时，需要关闭数据库：

```
>>> db.close()
```

## 14.7 封存

**dbm** 的限制之一是键和值都必须是字符串或字节。如果尝试使用其他类型，则会出现错误。

**pickle** 模块可以帮忙。它可以将几乎所有类型的对象转换为适合保存到数据库的字符串形式，并可以将字符串转换回来成为对象。

**pickle.dumps** 接收一个对象作为参数，并返回它的字符串表达式（**dumps** 是“dump string”的简写，意即转储字符串）：

```
>>> import pickle
>>> t = [1, 2, 3]
>>> pickle.dumps(t)
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

这个格式不适合人眼阅读；它是为了方便**pickle** 模块的转换而设计的。**pickle.loads**（load string，即加载字符串）重新构造对

象:

```
>>> t1 = [1, 2, 3]
>>> s = pickle.dumps(t1)
>>> t2 = pickle.loads(s)
>>> t2
[1, 2, 3]
```

虽然新的对象和旧有对象的值相同，但（通常来说）它们不是同一个对象:

```
>>> t1 == t2
True
>>> t1 is t2
False
```

也就是说，封存再解封，和复制对象效果相同。

你可以使用**pickle** 向数据库存储非字符串的值。事实上，这个组合如此常用，以至于**Python**已经将它们封装起来成为一个模块，叫作**shelve**。

## 14.8 管道

大部分操作系统都提供了命令行接口，也称为**字符界面**（**shell**）。字符界面通常会提供命令来浏览文件系统和启动应用程序。例如，在**Unix**中，可以使用**cd** 来更换目录，使用**ls** 来展示目录中的内容，以及打入**firefox** 来启动浏览器。

任何在字符界面能启动的程序都可以在Python中使用**管道对象**（**pipe object**）来启动。管道对象代表一个正在运行的程序。

例如，Unix命令**ls -l**以长格式展示当前目录的内容。可以使用**os.popen** [\[1\]](#)来启动**ls**：

```
>>> cmd = 'ls -l'
>>> fp = os.popen(cmd)
```

参数是一个字符串，它包含一个**shell**命令。返回值是一个和打开的文件差不多的对象。可以使用**readline**来逐行读取**ls**进程的输出，或者使用**read**一次读取所有输出：

```
>>> res = fp.read()
```

当你完成时，可以像文件一样关闭这个管道：

```
>>> stat = fp.close()
>>> print(stat)
None
```

返回值是**ls**进程的最终状态；**None**代表它正常结束了（没有错误）。

例如，大部分Unix系统都提供了一个叫作**md5sum**的命令，它读取文件的内容并计算出一个“校验和”（**checksum**）。你可以在<http://en.wikipedia.org/wiki/Md5>阅读MD5的相关信息。这个命令提供了一个高效的方法，用来对比两个文件是否包含相同的内容。不同的内

容生成相同的校验和的概率极低（也就是，在宇宙崩溃之前不大可能发生）。

可以在Python中使用管道来运行md5sum，并获得结果：

```
>>> filename = 'book.tex'
>>> cmd = 'md5sum ' + filename
>>> fp = os.popen(cmd)
>>> res = fp.read()
>>> stat = fp.close()
>>> print res
1e0033f0ed0656636de0d75144ba32e0  book.tex
>>> print(stat)
None
```

## 14.9 编写模块

任何包含Python代码的文件都可以作为模块导入。例如，如果你有一个文件wc.py，其代码如下：

```
def linecount(filename):
    count = 0
    for line in open(filename):
        count += 1
    return count

print(linecount('wc.py'))
```

如果你运行这个程序，它会读取自身的内容，并打印出文件的行数，即7。你也可以像这样导入它：

```
>>> import wc
7
```

现在你有一个模块对象`wc`了：

```
>>> wc
<module 'wc' from 'wc.py'>
```

该模块对象提供了`linecount`：

```
>>> wc.linecount('wc.py')
7
```

上述就是在Python中编写模块的方法。

这个例子唯一的问题是当你导入模块时，它会运行底部的测试代码。正常情况下，当你导入一个模块时，它会定义新的函数，但不会运行。

作为模块导入的程序，通常使用如下模式：

```
if __name__ == '__main__':
    print(linecount('wc.py'))
```

`__name__` 是一个内置变量，当程序启动时就会被设置。如果程序作为脚本执行，`__name__` 的值是 `'__main__'`；此时，测试代码会被运行。否则，如果程序作为模块被导入，则测试代码就被跳过了。

作为练习，把这个例子输入到一个文件`wc.py`中，并将它作为一个脚本运行。然后运行Python解释器，并导入`wc`。当模块被导入时，`__name__` 的值是什么？

警告：如果你导入一个已经被导入的模块，**Python**什么都不做。它不会重新读取文件，即使文件已经修改。

如果你想要重载一个模块，可以使用内置函数**reload**，但它也可能会有棘手的问题。所以最安全的办法是重启解释器，并再次导入模块。

## 14.10 调试

当你读取和写入文件时，可能会遇到和空白字符相关的问题。这些问题可能会很难调试，因为空格、制表符和换行符通常都是不可见的：

```
>>> s = '1 2\t 3\n 4'
>>> print(s)
1 2      3
 4
```

内置函数**repr**可以帮忙。它接收任何对象作为参数，并返回对象的字符串表达形式。对于字符串来说，它使用反斜杠序列来展示空白字符：

```
>>> print (repr(s))
'1 2\t 3\n 4'
```

这样可以帮助调试。

另一个你可能遇到的问题是不同的系统使用不同的字符表示换行。有的系统使用一个换行符，即**\n**。另外的系统使用一个回车符，

即`\r`。也有的系统两者都使用。如果你在不同的系统间移动文件，这些不一致之处可能会导致问题。

大多数系统都有程序可以将一种格式转换为另一种。你可以在 <http://en.wikipedia.org/wiki/Newline> 找到它们（并阅读这个问题的更多信息）。或者，当然，你也可以自己写一个。

## 14.11 术语表

**持久性（persistent）**：程序的一种属性，它会一直运行，并至少保存一部分数据在永久存储中。

**格式操作符（format operator）**：一个操作符，即`%`，它接收一个格式字符串，以及一个元组，并生成字符串，其中包括了元组的各个依据格式字符串里指定的方式格式化的元素。

**格式字符串（format string）**：一个字符串，被格式操作符所用，内部包含格式序列。

**格式序列（format sequence）**：格式字符串中出现的字符序列，如`%d`，它指定一个值如何格式化。

**文本文件（text file）**：存储在类似硬盘这样的永久存储中的字符串序列。

**目录（directory）**：有名称的文件集合。也称为文件夹。

**路径（path）**：用来标定一个文件的字符串。

相对路径（relative path）：从当前目录开始的路径。

绝对路径（absolute path）：从文件系统的顶级目录开始的路径。

捕获（catch）：使用`try`和`except`语句来阻止一个异常终止程序的行为。

数据库（database）：一个文件，其内容组织类似于字典，将键映射到值。

字节组对象（bytes object）：一个和字符串相似的对象。

命令行（shell）：一个程序，允许用户键入命令并通过调用其他程序来执行这些命令。

管道对象（pipe object）：代表一个运行中的程序的对象，让Python程序可以运行命令并读取结果。

## 14.12 练习

### 练习14-1

写一个名为`sed`的函数，接收如下参数：一个模式字符串，一个替换用字符串，以及两个文件名。它应该读取第一个文件，并将内容写入第二个文件（如果需要则新建它）。如果文件中任何地方出现了模式字符串，应该替换掉。



如果在打开、读取、写入或关闭文件的过程中遇到错误，你的程序应当能捕获异常，打印一个错误信息，并退出。

解答：<http://thinkpython2.com/code/sed.py>。

## 练习14-2

如果你从[http://thinkpython2.com/code/anagram\\_sets.py](http://thinkpython2.com/code/anagram_sets.py)下载我对练习12-2的解答，你会发现它创建一个字典，将一个排好序的字母串映射到可以由这些字母组成的单词的列表。例如，'opst' 映射到 ['opts', 'post', 'pots', 'spot', 'stop', 'tops'] 列表。

编写一个模块，导入 `anagram_sets`，并提供两个新函数：  
`store_anagrams` 应当存储回文字典到一个“shelf”中；  
`read_anagrams` 应当查询一个单词，并返回它的回文的列表。

解答：[http://thinkpython2.com/code/anagram\\_db.py](http://thinkpython2.com/code/anagram_db.py)。

## 练习14-3

在一个庞大的MP3文件的集合中，有可能同一首歌有多个副本，保存在不同的目录中，或者文件名不同。这个练习的目的是搜索重复的歌。

1. 编写一个程序递归搜索目录及其所有的子目录，并返回所有指定后缀（如 `.mp3`）的文件的完整路径的列表。提示：`os.path` 提供了几几个有用的方法来操纵文件和路径名称。

2. 要发现重复文件，需要使用`md5sum` 来计算每个文件的“校验和”。如果两个文件的校验和相同，它们很可能有相同的内容。

3. 你可以使用Unix命令`diff` 来复审检验。

解答: [http://thinkpython2.com/code/find\\_duplicates.py](http://thinkpython2.com/code/find_duplicates.py)

---

[1] `popen` 现在已经计划废止了，也就是说我们应当不再使用它，而是开始使用`subprocess` 模块。但对于简单的情形，我发现`subprocess` 过度复杂了。所以我仍然继续使用`popen`，直到它被完全废止。

## 第15章 类和对象

到现在你已经知道如何使用函数来组织代码，以及如何用内置类型来组织数据。下一步将学习“面向对象编程”，面向对象编程使用自定义的类型同时组织代码和数据。面向对象编程是一个很大的话题，需要好几章来讨论。

本章的代码示例可以从<http://thinkpython2.com/code/Point1.py>下载，练习的解答可以在[http://thinkpython2.com/code/Point1\\_soln.py](http://thinkpython2.com/code/Point1_soln.py)下载。

### 15.1 用户定义类型

我们已经使用了很多Python的内置类型；现在我们要定义一个新类型。作为示例，我们将会新建一个类型`Point`，用来表示二维空间中的一个点。

在数学的表示法中，点通常使用括号中逗号分隔两个坐标表示。例如， $(0, 0)$ 表示原点，而 $(x, y)$ 表示一个在原点右侧 $x$ 单位，上方 $y$ 单位的点。

在Python中，有好几种方法可以表达点。

- 我们可以将两个坐标分别保存到变量`x`和`y`中。
- 我们可以将坐标作为列表或元组的元素存储。

- 我们可以新建一个类型用对象表达点。

新建一个类型比其他方法更复杂一些，但它的优点很快就会显现出来。

用户定义的类型也称为**类**（**class**）。类的定义如下所示：

```
class Point:
    """Represents a point in 2-D space."""
```

定义头表示新的类名为**Point**。定义体是一个文档字符串，解释这个类的用途。可以在类定义中定义变量和函数，我们会在后面回到这个话题。

定义一个叫作**Point**的类会创建一个**对象类**（object class）。

```
>>> Point
<class '__main__.Point'>
```

因为**Point**是在程序顶层定义的，它的“全名”是**\_\_main\_\_.Point**。

类对象像一个创建对象的工厂。要新建一个**Point**对象，可以把**Point**当作函数来调用：

```
>>> blank = Point()
>>> blank
<__main__.Point object at 0xb7e9d3ac>
```

返回值是到一个**Point** 对象的引用，我们将它赋值给变量**blank**。

新建一个对象的过程称为**实例化**（**instantiation**），而对象是这个类的一个**实例**。

在打印一个实例时，**Python**会告诉你它所属的类型，以及存储在内存中的位置（前缀**0x** 表示后面的数字是十六进制的）。

每个对象都是某个类的实例，所以“对象”和“实例”这两个词很多情况下都可以互换。但是，在本章中我使用“实例”来表示一个自定义类型的对象。

## 15.2 属性

可以使用句点表示法来给实例赋值：

```
>>> blank.x = 3.0
>>> blank.y = 4.0
```

这个语法和从模块中选择变量的语法类似，如**math.pi** 或者 **string.whitespace**。但在这种情况下，我们是将值赋给一个对象的有命名的元素。这些元素称为**属性**（**attribute**）。

作为名词时，“**AT-trib-ute**”发音的重音在第一个音节，这与作为动词的“**a-TRIB-ute**”不同。

下面的图表展示了这些赋值的结果。展示一个对象和其属性的状态图称为**对象图**（object diagram），参见图15-1。

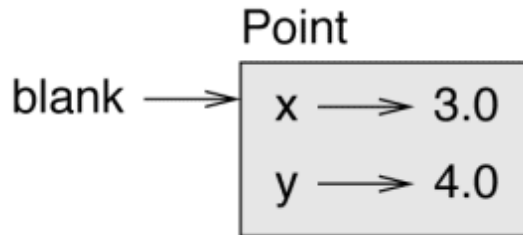


图15-1 对象图

变量**blank** 引用向一个**Point**对象，它包含了两个属性。每个属性引用一个浮点数。

可以使用相同的语法来读取一个属性的值：

```
>>> blank.y
4.0
>>> x = blank.x
>>> x
3.0
```

表达式**blank.x** 表示，“找到**blank** 引用的对象，并取得它的**x** 属性的值”。在这个例子中，我们将那个值赋值给一个变量**x**。变量**x** 和属性**x** 并不冲突。

可以在任意表达式中使用句点表示法。例如：

```
>>> '(%g, %g)' % (blank.x, blank.y)
'(3.0, 4.0)'
>>> distance = math.sqrt(blank.x**2 + blank.y**2)
>>> distance
5.0
```

可以将一个实例作为实参按通常的方式传递。例如：

```
def print_point(p):  
    print('(%g, %g)' % (p.x, p.y))
```

`print_point` 接收一个点作为形参，并按照数学表达式展示它。  
可以传入`blank` 作为实参来调用它：

```
>>> print_point(blank)  
(3.0, 4.0)
```

在函数中，`p` 是`blank` 的一个别名，所以如果函数修改了`p`，则`blank` 也会改变。

作为练习，编写一个叫作`distance_between_points` 的函数，接收两个`Point`对象作为形参，并返回它们之间的距离。

## 15.3 矩形

有时候对象应该有哪些属性非常明显，但也有时候需要你来做决定。例如，假设你在设计一个表达矩形的类。你会用什么属性来指定一个矩形的位置和尺寸呢？可以忽略角度，为了简单起见，假定矩形不是垂直的就是水平的。

最少有以下两种可能。

- 可以指定一个矩形的一个角落（或者中心点）、宽度以及高度。
- 可以指定两个相对的角落。

现在还很难说哪一种方案更好，所以作为示例，我们仅先实现第一个。

下面是这个类的定义：

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
```

文档字符串列出了属性：**width** 和 **height** 是数字；**corner** 是一个 **Point** 对象，用来指定左下角的顶点。

要表达一个矩形，需要实例化一个 **Rectangle** 对象，并对其属性赋值：

```
box = Rectangle()
box.width = 100.0
box.height = 200.0
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

表达式 **box.corner.x** 表示，“去往 **box** 引用的对象，并选择属性 **corner**；接着去往那个对象，并选择属性 **x**”。

图15-2展示了这个对象的状态。作为另一个对象的属性存在的对象是**内嵌**的。



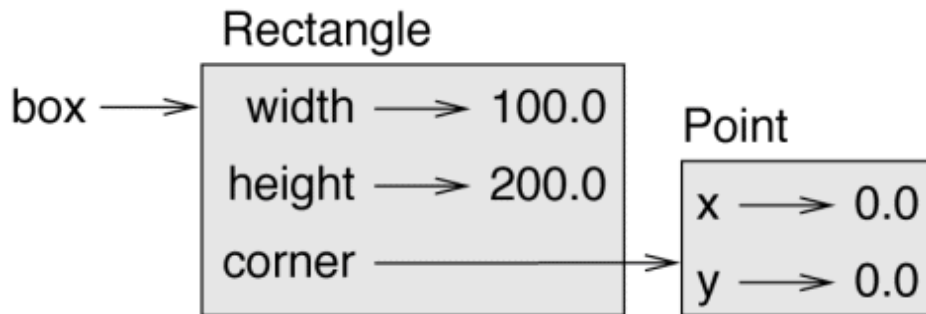


图15-2 对象图

## 15.4 作为返回值的实例

函数可以返回实例。例如，`find_center` 接收 `Rectangle` 对象作为参数，并返回一个 `Point` 对象，包含这个 `Rectangle` 的中心点的坐标：

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

下面是一个示例，传入 `box` 作为实参，并将结果的 `Point` 对象赋给变量 `center`：

```
>>> center = find_center(box)
>>> print_point(center)
(50, 100)
```

## 15.5 对象是可变的

可以通过给一个对象的某个属性赋值来修改它的状态。例如，要修改一个矩形的尺寸而保持它的位置不变，可以修改属性`width`和`height`的值：

```
box.width = box.width + 50
box.height = box.height + 100
```

也可以编写函数来修改对象。例如，`grow_rectangle` 接收一个 `Rectangle` 对象和两个数，`dwidth` 和 `dheight`，并把这些数加到矩形的宽度和高度上：

```
def grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

下面是展示这个函数效果的示例：

```
>>> box.width, box.height
(150.0, 300.0)
>>> grow_rectangle(box, 50, 100)
>>> box.width, box.height
(200.0, 400.0)
```

在函数中，`rect` 是 `box` 的别名，所以如果当修改了 `rect` 时，`box` 也改变。

作为练习，编写一个名为 `move_rectangle` 的函数，接收一个 `Rectangle` 对象和两个分别名为 `dx` 和 `dy` 的数值。它应当通过将 `dx` 添加到 `corner` 的 `x` 坐标和将 `dy` 添加到 `corner` 的 `y` 坐标来改变矩形的位置。

## 15.6 复制

别名的使用有时候会让程序更难阅读，因为一个地方的修改可能会给其他地方带来意想不到的变化。要跟踪掌握所有引用到一个给定对象的变量非常困难。

使用别名的常用替代方案是复制对象。`copy` 模块里有一个函数 `copy` 可以复制任何对象：

```
>>> p1 = Point()
>>> p1.x = 3.0
>>> p1.y = 4.0

>>> import copy
>>> p2 = copy.copy(p1)
```

`p1` 和 `p2` 包含相同的数据，但是它们不是同一个 `Point` 对象。

```
>>> print_point(p1)
(3, 4)
>>> print_point(p2)
(3, 4)
>>> p1 is p2
False
>>> p1 == p2
False
```

正如我们预料，`is` 操作符告诉我们 `p1` 和 `p2` 不是同一个对象。但你可能预料 `==` 能得到 `True` 值，因为这两个点包含相同的数据。如果那样，你会失望地发现对于实例来说，`==` 操作符的默认行为和 `is` 操作符相同，它会检查对象同一性，而不是对象相等性。这是因为对于用户自定义类型，`Python` 并不知道怎么才算相等。至少现在还不行。

如果使用`copy.copy` 复制一个`Rectangle`，你会发现它复制了`Rectangle`对象但并不复制内嵌的`Point`对象：

```
>>> box2 = copy.copy(box)
>>> box2 is box
False
>>> box2.corner is box.corner
True
```

图15-3展示了这个操作的对象图。这个操作称为**浅复制**（`shallow copy`），因为它复制对象及其包含的任何引用，但不复制内嵌对象。

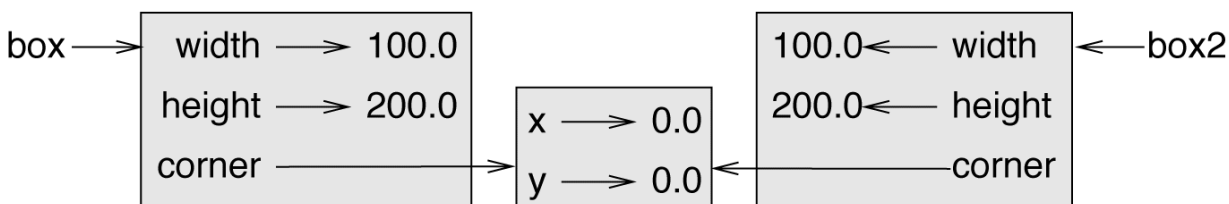


图15-3 对象图

对于大多数应用，这并不是你所想要的。在这个例子里，对一个`Rectangle`对象调用`grow_rectangle`并不会影响其他对象，但对任何一个`Rectangle`对象调用`move_rectangle`都会影响全部两个对象！这种行为既混乱不清，又容易导致错误。

幸好，`copy` 模块还提供了一个名为`deepcopy`的方法，它不但复制对象，还会复制对象中引用的对象，甚至它们引用的对象，依次类推。所以你并不会惊讶这个操作为何称为**深复制**（`deep copy`）。

```
>>> box3 = copy.deepcopy(box)
>>> box3 is box
False
>>> box3.corner is box.corner
False
```

---

`box3` 和 `box` 是两个完全分开的对象。

作为练习，编写 `move_rectangle` 的另一个版本，它会新建并返回一个 `Rectangle` 对象，而不是直接修改旧对象。

## 15.7 调试

开始操作对象时，可能会遇到一些新的异常。如果试图访问一个并不存在的属性，会得到 `AttributeError`：

```
>>> p = Point()
>>> p.x = 3
>>> p.y = 4
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

如果不清楚一个对象是什么类型，可以问：

```
>>> type(p)
<class '__main__.Point'>
```

也可以使用 `isinstance` 来检查对象是否是某个类的实例：

```
>>> isinstance(p, Point)
True
```

如果不确定一个对象是否拥有某个特定的属性，可以使用内置函数 `hasattr`：

```
>>> hasattr(p, 'x')
True
```

```
>>> hasattr(p, 'z')
False
```

第一个形参可以是任何对象，第二个形参是一个包含属性名称的字符串。

也可以使用`try`语句来尝试对象是否拥有你需要的属性：

```
try:
    x = p.x
except AttributeError:
    x = 0
```

这种方法可以使编写适用于不同类型的函数更加容易。关于这一主题的更多内容参见17.9节。

## 15.8 术语表

类（`class`）：一个用户定义的类型。类定义会新建一个类对象。

类对象（`class object`）：一个包含用户定义类型的信息的对象。类对象可以用来创建该类型的实例。

实例（`instance`）：属于某个类的一个对象。

实例化（`instanciate`）：创建一个新对象。

属性（`attribute`）：一个对象中关联的有命名的值。

内嵌对象（**embedded object**）：作为一个对象的属性存储的对象。

浅复制（**shallow copy**）：复制对象的内容，包括内嵌对象的引用；**copy** 模块中的**copy** 函数实现了这个功能。

深复制（**deep copy**）：复制对象的内容，也包括内嵌对象，以及它们内嵌的对象，依次类推；**copy** 模块中的**deepcopy** 函数实现了这个功能。

对象图（**object diagram**）：一个展示对象、对象的属性以及属性的值的图。

## 15.9 练习

### 练习15-1

定义一个新的名为**Circle** 的类表示圆形，它的属性有**center** 和**radius**，其中**center** 是一个**Point**对象，而**radius** 是一个数。

实例化一个**Circle**对象来代表一个圆心在(150, 100)、半径为75的圆形。

编写一个函数**point\_in\_circle**，接收一个**Circle**对象和一个**Point**对象，并当**Point**处于**Circle**的边界或其内时返回**True**。

编写一个函数**rect\_in\_circle**，接收一个**Circle**对象和一个**Rectangle**对象，并在**Rectangle**的任何一个角落在**Circle**之内时返回

True。另外，还有一个更难版本，需要在Rectangle的任何部分都落在圆圈之内时返回True。

解答：<http://thinkpython2.com/code/Circle.py>。

## 练习15-2

编写一个名为draw\_rect的函数，接收一个Turtle对象和一个Rectangle对象作为形参，并使用Turtle来绘制这个Rectangle。如何使用Turtle对象的示例参见第4章。

编写一个名为draw\_rect的函数，接收一个Turtle对象和一个Circle对象，并绘制出Circle。

解答：<http://thinkpython2.com/code/draw.py>。



## 第16章 类和函数

现在我们已经知道如何创建新的类型，下一步是编写接收用户定义对象作为参数或者将其当作结果返回的函数。本章我会展示“函数式编程风格”，以及两个新的程序开发计划。

本章的代码示例可以从<http://thinkpython2.com/code/Time1.py>下载。练习的解答在[http://thinkpython2.com/code/Time1\\_soln.py](http://thinkpython2.com/code/Time1_soln.py)。

### 16.1 时间

作为用户定义类型的另一个例子，我们定义一个叫作**Time** 的类，用于记录一天里的时间。类定义如下：

```
class Time:
    """Represents the time of day.

    attributes: hour, minute, second
    """
```

我们可以创建一个**Time** 对象并给其属性小时数、分钟数和秒钟数赋值：

```
time = Time()
time.hour = 11
time.minute = 59
time.second = 30
```

`Time` 对象的状态图参见图16-1。

作为练习，编写一个叫作`print_time`的函数，接收一个`Time`对象作为形参并以“`hour:minute:second`”的格式打印它。提示：格式序列'`%.2d`'可以以最少两个字符打印一个整数，如果需要，它会在前面添加前缀0。

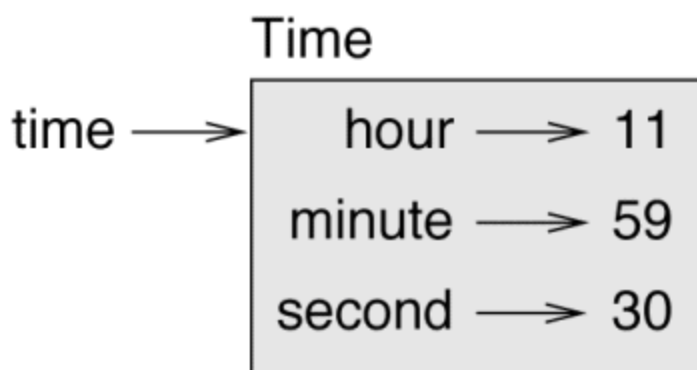


图16-1 对象图

编写一个布尔函数`is_after`，接收两个`Time`对象，`t1`和`t2`，并若`t1`在`t2`时间之后则返回`True`，否则返回`False`。挑战：不许使用`if`表达式。

## 16.2 纯函数

在下面几节中，我们会编写两个用来增加时间值的函数。它们展示了两种不同类型的函数：纯函数和修改器。它们也展示了我称为**原型和补丁**（`prototype and patch`）的开发计划。这是一种对应复杂问题的方法，从一个简单的原型开始，并逐渐解决更多的复杂情况。

下面是`add_time` 的一个简单原型:

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second
    return sum
```

这个函数创建一个新的`Time`对象，初始化它的属性，并返回这个新对象的一个引用。这被称为一个**纯函数**，因为它除了返回一个值之外，并不修改作为实参传入的任何对象，也没有任何如显示值或获得用户输入之类的副作用。

为了测试这个函数，我将创建两个`Time`对象: `start`，存放一个电影（如*Monty Python and the Holy Grail*）的开始时间；`duration`，存放电影的播放时间，在这里是1小时35分钟。

`add_time` 计算出电影何时结束。

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 0

>>> duration = Time()
>>> duration.hour = 1
>>> duration.minute = 35
>>> duration.second = 0

>>> done = add_time(start, duration)
>>> print_time(done)
10:80:00
```

结果**10:80:00** 可能并不是你所期望的。问题在于这个函数并没有处理好秒数或者分钟数超过60的情况。当此发生时，我们需要将多余的秒数“进位”到分钟数，将多余的分钟数“进位”到小时数。

下面是一个改进的版本：

```
def add_time(t1, t2):
    sum = Time()
    sum.hour = t1.hour + t2.hour
    sum.minute = t1.minute + t2.minute
    sum.second = t1.second + t2.second

    if sum.second >= 60:
        sum.second -= 60
        sum.minute += 1

    if sum.minute >= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```

虽然这个函数是正确的，它已经开始变大了。我们会在后面看到一个更短的版本。

## 16.3 修改器

有时候用函数修改传入的参数对象是很有用的。在这种情况下，修改对调用者是可见的。这样工作的函数称为**修改器**（**modifier**）。

函数**increment** 给一个**Time** 对象增加指定的秒数，可以自然地写为一个修改器。下面是一个初稿：

```
def increment(time, seconds):
    time.second += seconds
```

```
if time.second >= 60:
    time.second -= 60
    time.minute += 1

if time.minute >= 60:
    time.minute -= 60
    time.hour += 1
```

第一行进行基础操作；后面的代码处理我们前面看到的特殊情况。

这个函数正确吗？如果seconds比60大很多，会发生什么？

在那种情况下，只进位一次是不够的；我们需要重复进位，直到time.second比60小。一个办法是使用while语句替代if语句。那样会让函数变正确，但并不很高效。作为练习，编写正确的increment版本，并不包含任何循环。

任何可以使用修改器做到的功能都可以使用纯函数实现。事实上，有的编程语言只允许使用纯函数。有证据表明使用纯函数的程序比使用修改器的程序开发更快，错误更少。但有时候修改器还是很方便的，并且函数式程序的运行效率不那么高。

总的来说，我推荐你只要合理的时候，都尽量编写纯函数，而只有在有绝对说服力的原因时才使用修改器。这种方法可以称作**函数式编程风格**。

作为练习，编写一个increment的纯函数版本，创建并返回一个新的Time对象，而不是修改参数。

## 16.4 原型和计划

刚才我展示的开发计划称为“原型和补丁”。对每个函数，我编写一个可以进行基本计算的原型，再测试它，从中发现错误并打补丁。

这种方法在对问题的理解并不深入时尤其有效。但增量地修正可能会导致代码过度复杂（因为它们需要处理很多特殊情况），并且也不够可靠（因为很难知道你是否已经找到了所有错误）。

另一种方法是**有规划开发**（designed development）。对问题有更高阶的理解能够让编程简单得多。在上面的问题中，如果更深入地理解，可以发现Time对象实际上是六十进制数里的3位数（参见<http://en.wikipedia.org/wiki/Sexagesimal>）！second 属性是“个位数”，minute 属性是“60位数”，而hour 属性是“360位数”。

在编写add\_time 和increment 时，我们实际上是在六十进制上进行加减，因此才需要从一位进位到另一位。

这个观察让我们可以考虑整个问题的另一种解决方法——我们可以将Time对象转换为整数，并利用计算机知道如何做整数运算的事实。

下面是一个将Time对象转换为整数的函数：

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

而下面是一个将整数转换回Time对象的函数（记着divmod 函数将第一个参数除以第二个参数，并以元组的形式返回商和余数）：

```
def int_to_time(seconds):  
    time = Time()  
    minutes, time.second = divmod(seconds, 60)  
    time.hour, time.minute = divmod(minutes, 60)  
    return time
```

你可能需要思考一下，并运行一些测试，来说服自己这些函数是正确的。一种测试它们的方法是对很多x 值检查 `time_to_int(int_to_time(x)) == x`。这是一致性检验的一个例子。

一旦确认它们是正确的，就可以使用它们重写add\_time：

```
def add_time(t1, t2):  
    seconds = time_to_int(t1) + time_to_int(t2)  
    return int_to_time(seconds)
```

这个版本比最初版本短得多，并且也很容易检验。作为练习，使用time\_to\_int 和int\_to\_time 重写increment 函数。

从某个角度看，在六十进制和十进制之间来回转换比只处理时间更难。进制转换更加抽象；我们对时间值的直觉更好。

但如果我们将时间看作六十进制数，并做好了编写转换函数（time\_to\_int 和int\_to\_time）的先期投入，就能得到一个更短，更可读，也更可靠的函数。

它也让我们今后更容易添加功能。例如，假设将两个**Time**对象相减来获得它们之间的时间间隔。简单的做法是使用借位实现减法。而使用转换函数则更简单，且更容易正确。

讽刺的是，有时候把一个问题弄得更难（或者更通用）反而会让它更简单（因为会有更少的特殊情况以及更少的出错机会）。

## 16.5 调试

一个**Time**对象当**minute** 和**second** 的值在0到60之间（包含0但不包含60）以及**hour** 是正值时，是合法的。**hour** 和**minute** 应当是整数值，但我们也许需要允许**second** 拥有小数值。

这些需求称为**不变式**，因为它们应当总是为真。换句话说，如果它们不为真，则一定有什么地方出错了。

编写代码来检查不变式可以帮你探测错误并找寻它们的根源。例如，你可以写一个像**valid\_time** 这样的函数，接收**Time**对象，并在它违反了一个不变式时，返回**False**：

```
def valid_time(time):
    if time.hour < 0 or time.minute < 0 or time.second < 0:
        return False
    if time.minute >= 60 or time.second >= 60:
        return False
    return True
```

接着在每个函数的开头，可以检查参数，确保它们是有效的：



```
def add_time(t1, t2):
    if not valid_time(t1) or not valid_time(t2):
        raise ValueError('invalid Time object in add_time')
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

或者可以使用一个**assert** 语句。它会检查一个给定的不变式，并当检查失败时抛出异常：

```
def add_time(t1, t2):
    assert valid_time(t1) and valid_time(t2)
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

**assert** 语句很有用，因为它们区分了处理普通条件的代码和检查错误的代码。

## 16.6 术语表

原型和补丁（**prototype and patch**）：一种开发计划模式，先编写程序的粗略原型，并测试，在找到错误时更正。

有规划开发（**planned development**）：一种开发计划模式，先对问题有了高阶的深入理解，并且比增量开发或者原型开发有更多的规划。

纯函数（**pure function**）：不修改任何形参对象的函数。大部分纯函数都有返回值。

修改器（**modifier**）：修改一个或多个形参对象的函数。大部分修改器都不返回值，也就是返回**None**。

函数式编程风格（**functional programming style**）：一种编程设计风格，其中大部分函数都是纯函数。

不变式（**invariant**）：在程序的执行过程中应当总是为真的条件。

**assert** 语句（**assert statement**）：一种检查某个条件，如果检查失败则抛出异常的语句。

## 16.7 练习

本章中的代码示例可以从<http://thinkpython2.com/code/Time1.py>下载，这些练习的解答可以从[http://thinkpython2.com/code/Time1\\_soln.py](http://thinkpython2.com/code/Time1_soln.py)下载。

### 练习16-1

编写一个函数**mul\_time** 接收一个**Time**对象以及一个整数，返回一个新的**Time**对象，包含原始时间和整数的乘积。

然后使用**mul\_time** 来编写一个函数，接收一个**Time**对象表示一场赛车的结束时间，以及一个表示距离的数字，并返回一个**Time**对象表达平均节奏（每英里花费的时间）。

### 练习16-2

`datetime` 模块提供了`time` 对象，和本章中的`Time`对象类似，但它们提供了更丰富的方法和操作符。在 <http://docs.python.org/3/library/datetime.html> 阅读相关文档。

1. 使用`datetime` 模块来编写一个程序获取当前日期并打印出今天是周几。
2. 编写一个程序接收生日作为输入，并打印出用户的年龄，以及到他们下一次生日还需要的天数、小时数、分钟数和秒数。
3. 对于生于不同天的两个人，总有一天，一个人的年龄是另一个人的两倍。我们称这是他们的“双倍日”。编写一个程序接收两个生日，并计算出它们的“双倍日”。
4. 再增加一点挑战，编写一个更通用的版本，计算一个人比另一个人大 $n$  倍的日子。

解答: <http://thinkpython2.com/code/double.py>。

## 第17章 类和方法

虽然我们已经使用了Python的一些面向对象特性，但前两章的程序还算不上真正的面向对象，因为它们没有体现用户自定义类型之间的关联，以及操作它们的函数。下一步是将那些函数转换成方法，让这种关联更加明显。

本章的代码示例可以从<http://thinkpython2.com/code/Time2.py>下载，而本章练习的解答参见[http://thinkpython2.com/code/Point2\\_soln.py](http://thinkpython2.com/code/Point2_soln.py)。

### 17.1 面向对象特性

Python是一门**面向对象编程语言**，它提供了一些支持面向对象编程的语言特性，这些特性有如下明确的特征。

- 程序包括类定义和方法定义。
- 大部分计算都通过对象的操作来表达。
- 每个对象定义对应真实世界的某些对象或概念，而方法则对应真实世界中对象之间交互的方式。

例如，第16章中定义的**Time**类对应于人们记录一天中的时间的方式，而其中我们定义的函数对应于人们平时处理时间所做的事情。类似地，**Point**和**Rectangle**类对应于数学中点和矩形的概念。

目前为止，我们还没有利用上Python所提供的面向对象编程特性。严格地说，这些特性并不是必需的；它们中大部分都是我们已经做过的事情的另一种选择方案。但在很多情况下，这种方案更简洁，更能准确地表达程序的结构。

例如，在`Time1.py` 程序中，类定义和接着的函数定义并没有明显的关联。稍加观察，很明显每个函数都至少接收一个`Time`对象作为参数。

这种现象就是**方法**的由来。一个方法即是和某个特定类相关联的函数。我们已经见过字符串、列表、字典和元组的方法。本章中，我们会为用户定义类型定义方法。

方法和函数在语义上是一样的，但在语法上有两个区别。

- 方法定义写在类定义之中，更明确的表示类和方法的关联。
- 调用方法和调用函数的语法形式不同。

在接下来几节中，我们会将前两章中定义的函数转换为方法。这种转换是纯机械式的；你可以依照一系列步骤完成它。如果你能够轻松地在方法和函数之间转换，也就能够在任何情况下选择最适合的形式了。

## 17.2 打印对象

在第16章中，我们在练习16-1中定义了一个名为`Time`的类，你写过一个名为`print_time`的函数：

```
class Time:
    """Represents the time of day."""

def print_time(time):
    print('%.2d:%.2d:%.2d' % (time.hour, time.minute,
time.second))
```

要调用这个函数，需要传入一个**Time**对象作为实参：

```
>>> start = Time()
>>> start.hour = 9
>>> start.minute = 45
>>> start.second = 00
>>> print_time(start)
09:45:00
```

要把**print\_time** 转换为方法，我们只需要将函数定义移动到类定义中即可。注意缩进的变化。

```
class Time:
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute,
time.second))
```

现在有两种方式可以调用**print\_time**。第一种（更少见的）方式是使用函数调用语法：

```
>>> Time.print_time(start)
09:45:00
```

在这里的点表示法中，**Time** 是类的名称，而**print\_time** 是方法的名称。**start** 是作为参数传入的。

另一种（更简洁的）方式是使用方法调用语法：

```
>>> start.print_time()  
09:45:00
```

在这里的点表示法中，`print_time`（又一次）是方法的名称，而`start`是调用这个方法的对象，也称为**主体**（`subject`）。和一句话中主语用来表示这句话是关于什么东西的一样，方法调用的主体表示这个方法是关于哪个对象的。

在方法中，主体会被赋值给第一个形参，所以本例中`start`被赋值给`time`。

依惯例来，方法的第一个形参通常叫作`self`，所以`print_time`通常写成这样的形式：

```
class Time:  
    def print_time(self):  
        print('%.2d:%.2d:%.2d' % (self.hour, self.minute,  
self.second))
```

这种惯例的原因是一个隐喻。

- 函数调用的语法`print_time(start)`暗示函数是活动主体。它仿佛在说：“喂，`print_time`！这里是一个让你打印的对象。”
- 在面向对象编程中，对象是活动主体。类似`start.print_time()`的方法调用相当于说：“喂，`start`！请打印你自己。”

这种视角的改变可能变得更礼貌，但是否也更有用这一点却不那么明显。在我们已经见过的例子中，它也许并没有更有用。但有时候

将函数的责任转到对象上，使我们能够编写功能更丰富的函数（或方法），也使代码的维护和复用更容易。

作为练习，将16.4节中的函数`time_to_int` 重写为方法。你大概也会想将`int_to_time` 重写为方法，但这么做实际上没有什么意义，因为你找不到可以调用它的对象。

## 17.3 另一个示例

下面是函数`increment`（参见16.3节）的另一个重写成了方法的版本：

```
# inside class Time:

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

这个版本假设`time_to_int` 已经写成了方法。另外，注意它是一个纯函数，而不是一个修改器。

下面是调用`increment`的方式：

```
>>> start.print_time()
09:45:00
>>> end = start.increment(1337)
>>> end.print_time()
10:07:17
```



主体`start` 赋值给第一个形参`self`，实参`1337`，赋值给第二个形参`seconds`。

这种机制有时也会带来困惑，尤其在当程序出错的时候。例如，如果使用两个实参调用`increment`，则会得到：

```
>>> end = start.increment(1337, 460)
TypeError: increment() takes 2 positional arguments but 3 were
given
```

错误信息初看起来似乎很令人困惑，因为括号里只有两个实参。但调用的主体也被看作一个实参，所以其实总共有3个。

另外，**按位实参**（positional argument）指的是没有指定名称的实参，也就是说，它不是一个关键词实参。在下面这个函数调用中，`parrot` 和 `cage` 是按位实参，而 `dead` 是一个关键词实参：

```
sketch(parrot, cage, dead=True)
```

## 17.4 一个更复杂的示例

重写函数`is_after`（见16.1节）稍微更复杂一些，因为它接收两个`Time`对象作为形参。这种情形下，依惯例，第一个形参命名为`self`，而第二个形参命名为`other`：

```
# inside class Time:

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

要使用这个方法，需要在一个对象上调用它，并传入另一个对象作为实参：

```
>>> end.is_after(start)
True
```

这种语法的一个好处是，阅读起来几乎和英语一样：“end is after start?”。

## 17.5 `init` 方法

`init` 方法（即“initialization”的简写，意思是初始化）是一个特殊方法，当对象初始化时会被调用。它的全名是`__init__`（两个下划线，接着是`init`，再接着两个下划线）。`Time` 类的`init` 方法可能如下所示：

```
# inside class Time:

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

`__init__` 的形参和类的属性名称常常是相同的。语句

```
self.hour = hour
```

将形参`hour` 的值存储为`self` 的一个属性。

形参是可选的，所以当你不使用任何实参调用**Time** 时，会得到默认值：

```
>>> time = Time()
>>> time.print_time()
00:00:00
```

如果提供1个实参，它会覆盖**hour**：

```
>>> time = Time (9)
>>> time.print_time()
09:00:00
```

如果提供2个实参，它会覆盖**hour** 和**minute**：

```
>>> time = Time (9, 45)
>>> time.print_time()
09:45:00
```

如果提供3个实参，它们会覆盖全部3个默认值。

作为练习，为**Point** 类编写一个**init** 方法，接收**x** 和**y** 作为可选形参，并将它们的值赋到对应的属性上。

## 17.6 \_\_str\_\_ 方法

**\_\_str\_\_** 和**\_\_init\_\_** 类似，是一个特殊方法，它用来返回对象的字符串表达形式。

例如，下面是一个**Time**对象的**str** 方法：

```
# inside class Time:

    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute,
self.second)
```

当你打印对象时，Python会调用`str`方法。

```
>>> time = Time(9, 45)
>>> print(time)
09:45:00
```

当我编写一个新类时，我总是开始先写`__init__`，以便初始化对象，然后会写`__str__`，以便调试。

作为练习，为`Point`类编写一个`str`方法。创建一个`Point`对象并打印它。

## 17.7 操作符重载

通过定义其他的特殊方法，你可以为用户定义类型的各种操作符指定行为。例如，如果你为`Time`类定义一个`__add__`方法，则可以在`Time`对象上使用`+`操作符。

下面是这个方法的定义：

```
# inside class Time:

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

而下面是如何使用它：

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
```

当你对**Time**对象应用+操作符时，Python会调用**\_\_add\_\_**。当你打印结果时，Python会调用**\_\_str\_\_**。幕后其实发生了很多事情！

修改操作符的行为以便它能够作用于用户定义类型，这个过程称为**操作符重载**。对每一个操作符，Python都提供了一个对应的特殊方法，如**\_\_add\_\_**。更多细节，可以参见 <http://docs.python.org/3/reference/datamodel.html#specialnames>。

作为练习，为**Point**类编写一个**add**方法。

## 17.8 基于类型的分发

在前面一节中我们将两个**Time**对象相加，但你也可能会想要将一个**Time**对象加上一个整数。接下来是**\_\_add\_\_**的一个版本，检查**other**的类型，并调用**add\_time**或**increment**：

```
# inside class Time:

    def __add__(self, other):
        if isinstance(other, Time):
            return self.add_time(other)
        else:
            return self.increment(other)

    def add_time(self, other):
        seconds = self.time_to_int() + other.time_to_int()
```

```
        return int_to_time(seconds)

    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
```

内置函数`isinstance` 接收一个值与一个类对象，并当此值是此类的一个实例时返回`True`。

如果`other` 是一个`Time`对象，`__add__` 会调用`add_time`。否则它认为实参是整数，并调用`increment`。这个操作称为**基于类型的分发**（`type-based dispatch`），因为它根据形参的类型，将计算分发到不同的方法上。

下面是使用不同类型的实参调用`+` 操作符的示例：

```
>>> start = Time(9, 45)
>>> duration = Time(1, 35)
>>> print(start + duration)
11:20:00
>>> print(start + 1337)
10:07:17
```

遗憾的是，这个加法的实现并不满足交换律。如果整数是第一个操作数，则会得到：

```
>>> print(1337 + start)
TypeError: unsupported operand type(s) for +: 'int' and 'instance'
```

问题在于，这里和之前询问一个`Time`对象加上一个整数不同，`Python`在询问一个整数去加上一个`Time`对象，而它并不知道如何去做。但这个问题也有一个聪明的解决方案：特别方法`__radd__`，意

即“右加法”（right-side add）。当Time对象出现在+ 号的右侧时，会调用这个方法。下面是它的定义：

```
# inside class Time:

    def __radd__(self, other):
        return self.__add__(other)
```

而下面是如何使用：

```
>>> print(1337 + start)
10:07:17
```

作为练习，为Point类编写一个add 方法，可以接收一个Point对象或者一个元组。

- 如果第二个操作对象是一个Point对象，则方法应该返回一个新的Point对象，其x 坐标是两个操作对象的x 坐标的和，y 坐标也是类似。
- 如果第二个操作对象是一个元组，方法则将第一个元素和x 坐标相加，将第二个元素和y 坐标相加，并返回一个包含相加结果的新Point对象。

## 17.9 多态

当需要时，基于类型的分发很有用，但（幸运的是）我们并不总是需要它。通常可以编写函数处理不同类型的参数来避免它。

我们编写的很多处理字符串的函数，实际上对其他序列类型也可以用。例如，在11.1节中，我们使用**histogram**来记录单词中每个字母出现的次数：

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

这个函数对列表、元组甚至是字典都可用，只要**s**的元素是可散列的，因而可以用作**d**的键即可：

```
>>> t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
>>> histogram(t)
{'bacon': 1, 'egg': 1, 'spam': 4}
```

处理多个类型的函数称为**多态**（polymorphic）。多态可以促进代码复用。例如，用来计算一个序列所有元素的和的内置函数**sum**，对所有其元素支持加法的序列都可用。

由于**Time**对象提供了**add**方法，它们也可以使用**sum**：

```
>>> t1 = Time(7, 43)
>>> t2 = Time(7, 41)
>>> t3 = Time(7, 37)
>>> total = sum([t1, t2, t3])
>>> print (total)
23:01:00
```



总的来说，如果函数内部所有的操作都支持某种类型，那么这个函数就可以用于那种类型。

当你发现一个写好的函数，竟然有出人意料的效果，可以用于没有计划过的类型时，这才是最好的多态。

## 17.10 接口和实现

面向对象设计的目标之一是提高软件的可维护性，也就是说，当系统的其他部分改变时，程序还能够保持正确运行，并且能够修改程序来适应新的需求。

将接口和实现分离的设计理念，可以帮我们更容易达到这个目标。对于对象来说，那意味着类所提供的方法应该不依赖于其属性的表达方式。

例如，在本章中我们开发了一个类来表示一天中的时间。这个类提供的方法包括`time_to_int`、`is_after`和`add_time`。

我们可以使用几种不同的方式来实现这些方法。实现的细节依赖于我们表达时间概念的方式。在本章中，`Time`对象的属性是`hour`、`minute`和`second`。

用另一种方案，我们可以将这些属性替换成一个整数，表示从凌晨开始到现在的秒数。这种实现可能会让一些方法，如`is_after`，更容易实现，但也会让另一些方法更难实现。

在部署一个新类时，你可能会发现更好的实现。如果程序中其他部分用到你的类，则修改接口会非常消耗时间，并且容易产生错误。

但是，如果很谨慎小心地设计接口，则可以在不修改接口的情况下修改实现，这样程序的其他部分就不需要跟着修改。

## 17.11 调试

在程序运行的任何时刻，往对象上添加属性都是合法的，但如果遵守更严格的类型理论，让对象拥有相同的类型却有不同属性组，会很容易导致错误。通常来说，在`init`方法中初始化对象的全部属性是个好习惯。

如果并不清楚一个对象是否拥有某个属性，可以使用内置函数`hasattr`（参见15.7节）。

另一种访问一个对象的属性的方法是使用内置函数`vars`，它接收一个对象，并返回一个将属性名称（字符串形式）映射到属性值的字典对象：

```
>>> p = Point(3, 4)
>>> vars(p)
{'y': 4, 'x': 3}
```

为了调试，你可能会发现将这个函数放在手边是很有用的：

```
def print_attributes(obj):
    for attr in vars(obj):
        print (attr, getattr(obj, attr))
```

`print_attributes` 遍历对象的属性字典，并打印出每个属性的名称和相应的值。

内置函数`getattr` 接收一个对象以及一个属性名称（字符串形式）并返回属性的值。

## 17.12 术语表

面向对象语言（**object-oriented language**）：一种提供诸如用户定义类型和方法之类的语言特性，以方便面向对象编程的语言。

面向对象编程（**object-oriented programming**）：一种编程风格，数据和修改数据的操作组织成类和方法的形式。

方法（**method**）：在类定义之内定义的函数，在类的实例上调用。

主体（**subject**）：调用方法所在的对象。

按位实参（**positional argument**）：一个不包含参数名字的实参，所以它不是一个关键词实参。

操作符重载（**operator overloading**）：修改一个类似+号这样的操作符的行为，使之可以用于用户定义类型。

基于类型的分发（**type-based dispatch**）：一种编程模式，检查操作对象的类型，并对不同类型调用不同的函数。

多态（polymorphic）：函数的一种属性，可以处理多种类型的参数。

信息隐藏（information hiding）：对象提供的接口不应当依赖于其实现，特别是其属性的表达形式的原则。

## 17.13 练习

### 练习17-1

从<http://thinkpython2.com/code/Time2.py> 下载本章的代码。将 `Time` 的属性改为从凌晨开始到现在的秒数。接着修改方法（以及函数 `int_to_time`），以适应新的属性实现。你应该不需要修改 `main` 里面的测试代码。当你做完之后，输出应该和以前一样。

解答：[http://thinkpython2.com/code/Time2\\_soln.py](http://thinkpython2.com/code/Time2_soln.py)

### 练习17-2

这个练习提醒你关于Python的一种最常见且最难查找的错误的故事。

编写一个叫作 `Kangaroo`（袋鼠）的类，有如下方法。

1. 一个 `__init__` 方法，将属性 `pouch_contents`（口袋中的东西）初始化为一个空列表。

2. 一个`put_in_pouch`方法，接收任何类型的对象，并将它添加到`pouch_contents`中。

3. 一个`__str__`方法，返回`Kangaroo`对象以及口袋中的内容的字符串表达形式。

创建两个`Kangaroo`对象，将它们赋值到变量`kanga`和`roo`，并将`roo`添加到`kanga`的口袋中。

下载<http://thinkpython.com/code/BadKangaroo.py>，它包含了前面问题的解答，但里面有一个很大很丑陋的bug。找出并修复这个bug。

如果你遇到阻碍，可以下载<http://thinkpython.com/code/GoodKangaroo.py>，它解释了问题的原因，并提供了一个解决方案。

## 第18章 继承

和面向对象编程最常相关的语言特性就是继承（**inheritance**）。继承指的是根据一个现有的类型，定义一个修改版本的新类的能力。本章中我会使用几个类来表达扑克牌、牌组以及扑克牌型，用于展示继承特性。

如果你不玩扑克，可以在<http://en.wikipedia.org/wiki/Poker>里阅读相关介绍，但其实并不必要；我会在书中介绍练习中所需知道的东西。

本章的代码示例可以从<http://thinkpython2.com/code/Card.py>下载。

### 18.1 卡片对象

一副牌里有52张牌，共有4个花色，每种花色13张，大小各不相同。花色有黑桃（**Spade**）、红桃（**Heart**）、方片（**Diamond**）和草花（**Club**）（在桥牌中，这几个花色是降序排列的）。每种花色的13张牌分别为：**Ace**、2、3、4、5、6、7、8、9、10、**Jack**、**Queen**和**King**。根据你玩的不同游戏，**Ace**可能比**King**大，也可能比2小。

如果我们定义一个新对象来表示卡牌，则其属性显然应该是**rank**（大小）和**suit**（花色）。但属性的值就不那么直观了。一种可能是使用字符串，例如，用'**Spade**'表示花色，用'**Queen**'表示大小。这种实现的问题之一是比较大小和花色的高低时会比较困难。

另一种方案是使用整数来给大小和花色**编码**。在这个语境中，“编码”意味着我们要定义一个数字到花色，或者数字到大小的映射。这种编码并不意味着它是秘密（那样就应该称为“加密”了）。

例如，下表展示了花色和对应的整数编码：

Spades	→	3
Hearts	→	2
Diamonds	→	1
Clubs	→	0

这个编码令我们可以很容易地比较卡牌；因为更大的花色映射到更大的数字上，我们可以直接使用编码来比较花色。

卡牌大小的映射相当明显；每个数字形式的大小映射到相应的整数上，而对于花牌：

Jack	→	11
Queen	→	12
King	→	13

我使用“→”符号，是为了说明这些映射并不是Python程序的一部分。它们是程序设计的一部分，但并不在代码中直接表现。

**Card** 类的定义如下：

```
class Card:
    """Represents a standard playing card."""

    def __init__(self, suit=0, rank=2):
        self.suit = suit
        self.rank = rank
```

和前面一样，`init` 方法对每个属性定义一个可选形参。默认的卡牌是草花2。

要创建一个`Card`对象，使用你想要的花色和大小调用`Card`：

```
queen_of_diamonds = Card(1, 12)
```

## 18.2 类属性

为了能将`Card`对象打印成人们容易阅读的格式，我们需要将整数编码映射成对应的大小和花色。自然的做法是使用字符串列表。我们将这些列表赋到**类属性**上：

```
# 在Card类里：

suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
               '8', '9', '10', 'Jack', 'Queen', 'King']

def __str__(self):
    return '%s of %s' % (Card.rank_names[self.rank],
                          Card.suit_names[self.suit])
```

`suit_names` 和 `rank_names` 这样的变量，定义在类之中，但在任何方法之外，我们称为类属性。因为它们是和类对象`Card`相关联的。

这个术语和`suit` 与 `rank` 之类的变量相区别。那些称为**实例属性**，因为它们是和特定的实例相关联的。



两种属性都使用句点表示法访问。例如，在\_\_str\_\_中，`self` 是一个Card对象，而`self.rank` 是它的大小。相似地，`Card` 是一个类对象，而`Card.rank_names` 是关联到这个类的一个字符串列表。

每个卡片都有它自己的`suit` 和`rank`，但总共只有一个`suit_names` 和`rank_names`。

综合起来，表达式`Card.rank_names[self.rank]` 意思是“使用对象`self` 的属性`rank` 作为索引，从类`Card` 的列表`rank_names` 中选择对应的字符串”。

`rank_names` 的第一个元素是`None`，因为没有大小为0的卡牌。因为使用`None` 占据了一个位置，我们就可以得到从下标2到字符串'2' 这样整齐的映射。如果要避免这种操作，可以使用字典而不是列表。

利用现有的方法，可以创建并打印卡牌：

```
>>> card1 = Card(2, 11)
>>> print(card1)
Jack of Hearts
```

图18-1展示了`Card` 类对象和一个Card实例。`Card` 是一个类对象，所以它的类型是`type`。`card1` 的类型是`Card`。为了节省空间，我没有画出`suit_names` 和`rank_names` 的内容。

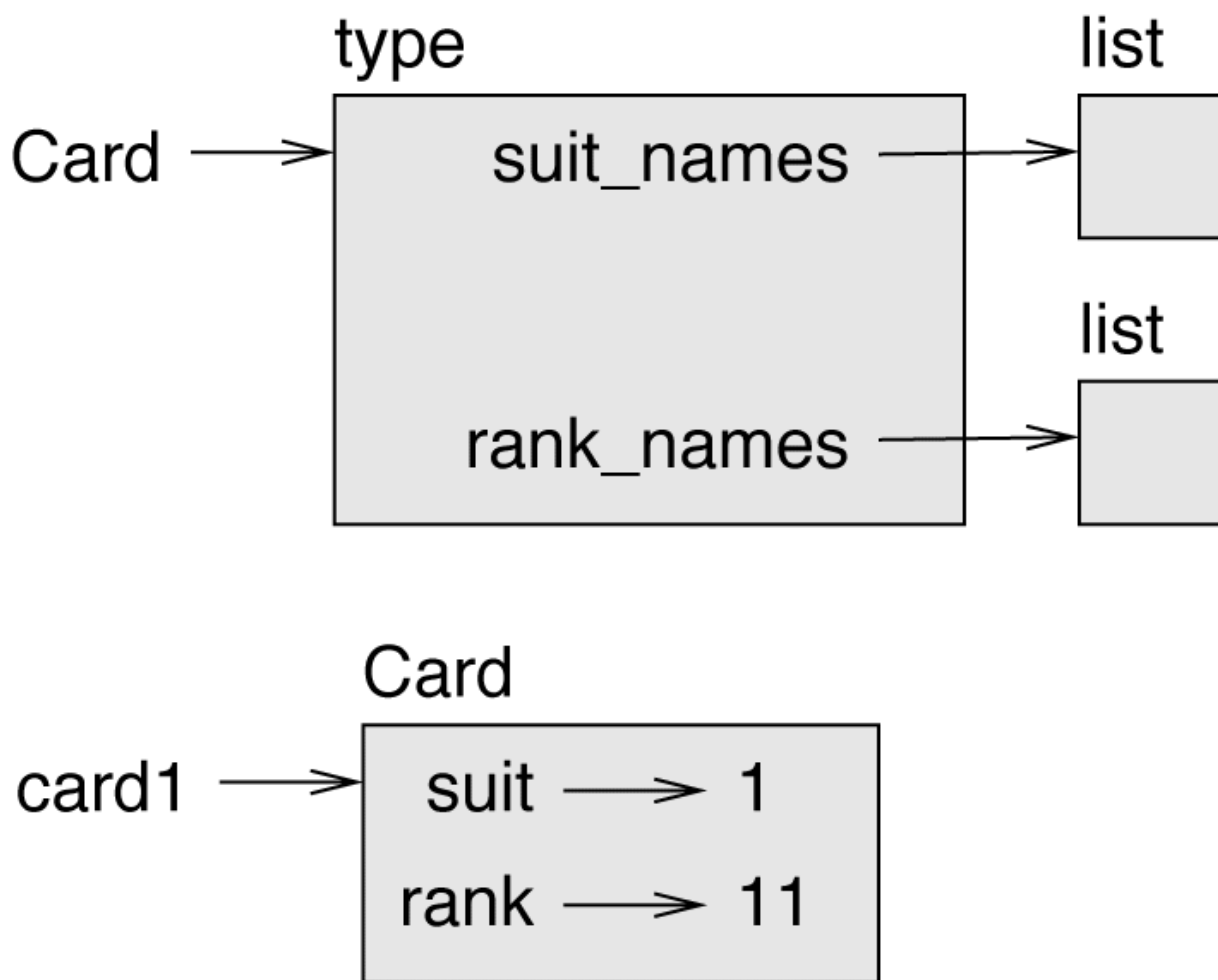


图18-1 对象图

### 18.3 对比卡牌

对于内置类型，我们用比较操作符（<、>、== 等）来比较对象并决定哪一个更大、更小或者相等。对于用户定义类型，我们可以通过提供一个方法 `__lt__`，代表“less than”，来重载内置操作符的行为。

`__lt__` 接收两个形参，`self` 和 `other`，当第一个对象严格小于第二个对象时返回 `True`。

卡牌的正确顺序并不显而易见。例如，草花3和方片2哪个更大？一个牌面数大，另一个花色大。为了比较卡牌，需要决定大小和花色哪个更重要。

这个问题的答案取决于你在玩哪种牌类游戏，但为了简单起见，我们随意做一个决定，认为花色更重要，于是，所有的黑桃比所有的方片都大，依此类推。

这一点决定后，我们就可以编写\_\_lt\_\_函数：

```
# 在 Card类里：

def __lt__(self, other):
    # 检查花色
    if self.suit < other.suit: return True
    if self.suit > other.suit: return False

    # 花色相同，检查大小
    return self.rank < other.rank
```

使用元组比较，可以写得更紧凑：

```
# 在Card类里：

def __lt__(self, other):
    t1 = self.suit, self.rank
    t2 = other.suit, other.rank
    return t1 < t2
```

作为练习，为时间对象编写一个\_\_lt\_\_方法。你可以使用元组比较，也可以考虑使用整数比较。

## 18.4 牌组

现在我们已经有了卡牌（card），下一步就是定义牌组（deck）。由于牌组是由卡牌组成的，很自然地，每个Deck对象应该有一个属性包含卡牌的列表。

下面是Deck的类定义。init方法创建cards属性，并生成52张牌的标准牌组：

```
class Deck:

    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                card = Card(suit, rank)
                self.cards.append(card)
```

填充牌组最简单的办法是使用嵌套循环。外层循环从0到3遍历各个花色。内层循环从1到13遍历卡牌大小。每次迭代使用当前的花色和大小创建一个新的Card对象，并将它添加到self.cards中。

## 18.5 打印牌组

下面是Deck的一个\_\_str\_\_方法：

```
# 在 Deck类里：

def __str__(self):
    res=[]
    for card in self.cards:
        res.append(str(card))
    return '\n'.join(res)
```

这个方法展示了一种累积构建大字符串的方法：先构建一个字符串的列表，再使用字符串方法`join`。内置函数`str`会对每个卡牌对象调用`__str__`方法并返回字符串表达形式。

由于我们对一个换行符调用`join`函数，卡片之间用换行分隔。下面是打印的结果：

```
>>> deck = Deck()
>>> print(deck)
Ace of Clubs
2 of Clubs
3 of Clubs
...
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
```

虽然结果显示了52行，它仍然是一个包含换行符的字符串。

## 18.6 添加、删除、洗牌和排序

为了能够发牌，我们需要一个方法从牌组中抽取一张牌并返回。列表方法`pop`为此提供了一个方便的功能：

```
# 在 Deck类里：

def pop_card(self):
    return self.cards.pop()
```

由于`pop`从列表中抽出最后一张牌，我们其实是从牌组的底端发牌的。

要添加一个卡牌，我们可以使用列表方法`append`：

```
# 在 Deck类里：

def add_card(self, card):
    self.cards.append(card)
```

像这样调用另一个方法，却不做其他更多工作的方法，有时候称为一个**饰面**（**veneer**）。这个比喻来自于木工行业，在木工行业中饰面是为了改善外观而粘贴到便宜的木料表面的薄薄的一层优质木料。

在这个例子里，`add_card` 是一个“薄薄”的方法，用更适合牌组的术语来表达一个列表操作。它改善了实现的外观（或接口）。

作为另一个示例，我们可以使用`random` 模块的函数`shuffle` 来编写一个`Deck`方法`shuffle`（洗牌）：

```
# 在 Deck类里：

def shuffle(self):
    random.shuffle(self.cards)
```

不要忘记导入`random` 模块。

作为练习，编写一个`Deck`方法`sort`，使用列表方法`sort` 来对一个`Deck` 中的卡牌进行排序。`sort` 使用我们定义的`__lt__`方法来决定顺序。

## 18.7 继承

继承是一种能够定义一个新类对现有的某个类稍作修改的语言特性。作为示例，假设我们想要一个类来表达一副“手牌”，即玩家手握的一副牌。一副手牌和一套牌组相似：都是由卡牌的集合组成，并且都需要诸如增加和移除卡牌的操作。

一副手牌和一套牌组也有区别；我们期望手牌拥有的一些操作，对牌组来说并无意义。例如，在扑克牌中，我们可能想要比较两副手牌来判断谁获胜了。在桥牌中，我们可能需要为一副手牌计算分数以叫牌。

这种类之间的关系——相似，但不相同——让它成为继承。要定义一个继承现有类的新类，可以把现有类的名称放在括号之中：

```
class Hand(Deck):  
    """Represents a hand of playing cards."""
```

这个定义说明**Hand** 从**Deck** 继承而来；这意味着我们可以像**Deck** 对象那样在**Hand**对象上使用**pop\_card** 和**add\_card** 方法。

当新类继承现有类时，现有的类被称为**父类**（**parent**），而新类则称为**子类**（**child**）。

在本例中，**Hand** 也会继承**Deck** 的**\_\_init\_\_** 方法，但它和我们想要的并不一样：我们不需要填充52张卡牌，**Hand**的**init** 方法应当初始化**cards** 为一个空列表。

如果我们为**Hand** 类提供一个**init** 方法，它会覆盖**Deck** 类的方法：

```
# 在Hand类里：

def __init__(self, lable=''):
    self.cards = []
    self.label = lable
```

在创建Hand对象时，Python会调用这个init 方法而不是Deck 中的那个：

```
>>> hand = Hand('new hand')
>>> hand.cards
[]
>>> hand.label
'new hand'
```

其他的方法是从Deck 中继承而来的，所以我们可以使用pop\_card 和add\_card 来出牌：

```
>>> deck = Deck()
>>> card = deck.pop_card()
>>> hand.add_card(card)
>>> print(hand)
King of Spades
```

下一步很自然地就是将这段代码封装起来成为一个方法move\_cards：

```
# 在 Deck类里：

def move_cards(self, hand, num):
    for i in range(num):
        hand.add_card(self.pop_card())
```



`move_cards` 接收两个参数，一个`Hand`对象以及需要出牌的牌数。它会修改`self` 和`hand`，返回`None`。

有的情况下，卡牌会从一副手牌中移除转入到另一幅手牌中，或者从手牌中回到牌组。你可以使用`move_cards` 来处理全部这些操作：`self` 既可以是一个`Deck`对象，也可以是一个`Hand`对象。而`hand`参数，虽然名字是`hand`，却也可以是一个`Deck`对象。

继承是很有用的语言特性。有些程序不用继承写，会有很多重复代码，使用继承后就会更加优雅。继承也能促进代码复用，因为你可以不修改父类的前提下对它的行为进行定制化。有的情况下，继承结构反映了问题的自然结构，所以也让设计更容易理解。

但另一方面，继承也可能会让代码更难读。有时候当一个方法被调用时，并不清楚到哪里能找到它的定义。相关的代码可能散布在几个不同的模块中。并且，很多可以用继承实现的功能，也能不用它实现，甚至可以实现得更好。

## 18.8 类图

至此我们已见过用于显示程序状态的栈图，以及用于显示对象的属性和属性值的对象图。这些图表展示了程序运行中的一个快照，所以当程序继续运行时它们会跟着改变。

它们也极其详细；在某些情况下，是过于详细了。而类图对程序结构的展示相对来说更加抽象。它不会具体显示每个对象，而是显示各个类以及它们之间的关联。

类之间有下面几种关联。

- 一个类的对象可能包含其他类的对象的引用。例如，每个 **Rectangle** 对象都包含一个到 **Point** 对象的引用，而每一个 **Deck** 对象包含到很多 **Card** 对象的引用。这种关联称为 **HAS-A**（有一个），也就是说，“矩形（**Rectangle**）中有一个点（**Point**）”。
- 一个类可能继承自另一个类。这种关系称为 **IS-A**（是一个），也就是说，“一副手牌（**Hand**）是一个牌组（**Deck**）”。
- 一个类可能依赖于另一个类，也就是说，一个类的对象接收另一个类的对象作为参数，或者使用另一个类的对象来进行某种计算。这种关系称为 **依赖**（**dependency**）。

**类图** 用图形展示了这些关系。例如，图18-2展示了 **Card**、**Deck** 和 **Hand** 之间的关系。

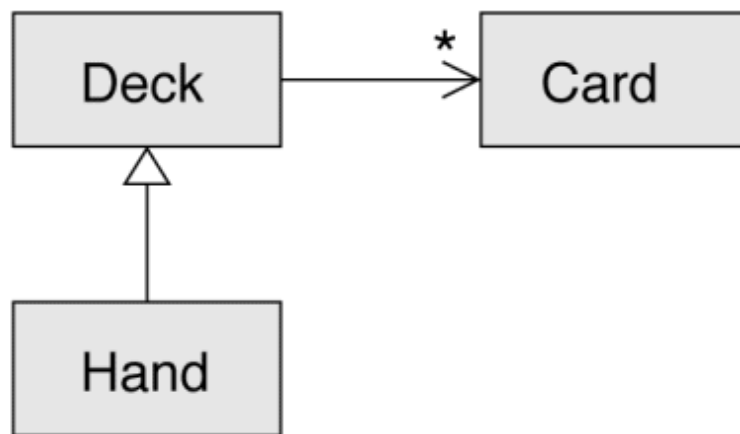


图18-2 类图

空心三角形箭头的线代表着一个 **IS-A** 关系；这里表示 **Hand** 是继承自 **Deck** 的。

标准的箭头表示HAS-A关系；这里表示Deck对象中有到Card对象的引用。

箭头附近的星号（\*）表示是**关联重数标记**；它表示Deck中有多少Cards。这个数可以是一个简单的数字，如52，或者一个范围，如5..7，或者一个星号，表示Deck可以有任意数量的Card引用。

图18-2中没有任何依赖关系。依赖关通常使用虚线箭头表示。或者，如果有太多的依赖，有时候会忽略它们。

更详细的图可能会显示出Deck对象实际上包含了一个Card的列表。但在类图中，像列表、字典这样的内置类型通常是不显示的。

## 18.9 数据封装

前几章展示了一个我们可以称为“面向对象设计”的开发计划。我们发现需要的对象，如Point、Rectangle和Time，并定义类来表达它们。每个类都是一个对象到现实世界（或者最少是数学世界）中的某种实体的明显对应。

但有时候你到底需要哪些对象、它们如何交互，并不那么显而易见。这时候你需要另一种开发计划。和之前我们通过封装和泛化来发现函数接口的方式相同，我们可以通过**数据封装**来发现类的接口。

13.8节提供了一个很好的示例。如果从<http://thinkpython2.com/code/markov.py>下载我的代码，你会发现它使用

了两个全局变量（`suffix_map` 和 `prefix`）并且在多个函数中进行读写。

```
suffix_map = {}  
prefix = ()
```

因为这些变量是全局的，我们每次只能运行一个分析。如果我们读入两个文本，它们的前缀和后缀就会添加到相同的数据结构中（最后可以用来产生一些有趣的文本）。

若要多次运行分析，并保证它们之间的独立，我们可以将每次分析的状态信息封装成一个对象。下面是它的样子：

```
class Markov:  
  
    def __init__(self):  
        self.suffix_map = {}  
        self.prefix = ()
```

接下来，我们将那些函数转换为方法。例如，下面是 `process_word`：

```
def process_word(self, word, order=2):  
    if len(self.prefix) < order:  
        self.prefix += (word,)   
        return  
  
    try:  
        self.suffix_map[self.prefix].append(word)  
    except KeyError:  
        # 如果这个前缀不存在，创建一项  
        self.suffix_map[self.prefix] = [word]  
  
    self.prefix = shift(self.prefix, word)
```

像这样转换程序——修改设计但不修改其行为——是重构（参见4.7节）的另一个示例。

这个例子给出了一个设计对象和方法的开发计划。

1. 从编写函数、（如果需要的话）读写全局变量开始。
2. 一旦你的程序能够正确运行，查看全局变量与使用它们的函数的关联。
3. 将相关的变量封装成为对象的属性。
4. 将相关的函数转换为这个新类的方法。

作为练习，从<http://thinkpython2.com/code/markov.py>下载我的Markov代码，并按照上面描述的步骤将全局变量封装为一个叫作Markov的新类的属性。

解答：<http://thinkpython2.com/code/Markov.py>（注意M是大写的）。

## 18.10 调试

继承会给调试带来新的挑战，因为当你调用对象的方法时，可能无法知道调用的到底是哪个方法。

假设你在编写一个操作Hand对象的函数。你可能希望能够处理所有类型的Hand，如PokerHands、BridgeHands等。如果你调用一个方

法，如`shuffle`，可能调用的是`Deck`中定义的方法，但如果任何子类重载了这个方法，则你调用的会是那个重载的版本。

一旦你无法确认程序的运行流程，最简单的解决办法是在相关的方法开头添加一个打印语句。如果`Deck.shuffle`打印一句`Running Deck.shuffle`这样的信息，则当程序运行时跟踪运行的流程。

或者，你也可以使用下面这个函数。它接收一个对象和一个方法名（字符串形式），并返回提供这个方法的定义的类：

```
def find_defining_class(obj, meth_name):
    for ty in type(obj).mro():
        if meth_name in ty.__dict__:
            return ty
```

下面是使用的示例：

```
>>> hand = Hand()
>>> find_defining_class(hand, 'shuffle')
<class 'Card.Deck'>
```

所以这个`Hand`对象的`shuffle`方法是在`Deck`类中定义的那个。

`find_defining_class`使用`mro`方法来获得用于搜索调用方法的类对象（类型）列表。“MRO”意思是“method resolution order”（方法查找顺序），是`Python`解析方法名称的时候搜索的类的顺序。

一个设计建议：每次重载一个方法时，新方法的接口应当和旧方法的一致。它应当接收相同的参数，返回相同的类型，并服从同样的前置条件与后置条件。如果遵循这个规则，你会发现任何为如`Deck`这

样的父类设计的函数，都可以使用**Hand**或**PokerHand**这样的子类的实例。

如果你破坏这个也称为“**Liskov**替代原则”的规则，你的代码可能会像一堆（不好意思）纸牌屋一样崩塌。

## 18.11 术语表

**编码（encode）**：使用一个集合的值来表示另一个集合的值，需要在它们之间建立映射。

**类属性（class attribute）**：关联到类对象上的属性。类属性定义在类定义之中，但在所有方法定义之外。

**实例属性（instance attribute）**：和类的实例关联的属性。

**饰面（veneer）**：一个方法或函数，它调用另一个函数，却不做其他计算，只是为了提供不同的接口。

**继承（inheritance）**：可以定义一个新类，它是一个现有的类的修改版本。

**父类（parent class）**：被子类所继承的类。

**子类（child class）**：通过继承一个现有的类来创建的新类，也叫作“subclass”。

**IS-A关联（IS-A relationship）**：子类与父类之间的关联。

**HAS-A关联**（HAS-A relationship）：两个类之间的一种关联：一个类包含另一个类的对象的引用。

**依赖**（dependency）：两个类之间的一种关联。一个类的实例使用另一个类的实例，但不把它们作为属性存储起来。

**类图**（class diagram）：用来展示程序中的类以及它们之间的关联的图。

**重数**（multiplicity）：类图中的一种标记方法，对于HAS-A关联，用来表示一个类中有多少对另一个类的对象的引用。

**数据封装**（data encapsulation）：一个程序开发计划。先使用全局变量来进行原型设计，然后将全局变量转换为实例属性做出最终版本。

## 18.12 练习

### 练习18-1

针对下面的程序，画一张UML类图，展示这些类以及它们之间的关联：

```
class PingPongParent:
    pass

class Ping(PingPongParent):
    def __init__(self, pong):
        self.pong = pong

class Pong(PingPongParent):
    def __init__(self, pings=None):
        if pings is None:
```



```
        self.pings = []
    else:
        self.pings = pings

    def add_ping(self, ping):
        self.pings.append(ping)

pong = Pong()
ping = Ping(pong)
pong.add_ping(ping)
```

## 练习18-2

编写一个名为**deal\_hands**的**Deck**方法，接收两个形参：手牌的数量以及每副手牌的牌数。它会根据形参创建新的**Hand**对象，按照每副手牌的牌数出牌，并返回一个**Hand**对象列表。

## 练习18-3

下面列出的是扑克牌中可能的手牌，按照牌值大小的增序（也是可能性的降序）排列。

- 对子（**pair**）：两张牌大小相同。
- 两对（**two pair**）：两个对子。
- 三条（**three of a kind**）：三张牌大小相同。
- 顺子（**straight**）：五张大小相连的牌（**Ace**既可以是最大也可以是最小，所以**Ace-2-3-4-5**是顺子，**10-Jack-Queen-King-Ace**也是，但**Queen-King-Ace-2-3**不是）。
- 同花（**flush**）：五张牌花色相同。
- 满堂红（**full house**）：三张牌大小相同，另外两张牌大小相同。
- 四条（**four of a kind**）：四张牌大小相同。

- 同花顺 (**straight flush**)：顺子（如上面的定义）里的五张牌都是花色相同的。

本练习的目标是预测这些手牌的出牌概率。

1. 从<http://thinkpython2.com/code>下载这些文件。

- **Card.py**：本章中介绍的**Card**、**Deck** 和**Hand** 类的完整代码。
- **PokerHand.py**：表达扑克手牌的一个类，实现并不完整，包含一些测试它的代码。

2. 如果你运行**PokerHand.py**，它会连出7组包含7张卡片的扑克手牌，并检查其中有没有顺子。在继续之前请仔细阅读代码。

3. 在**PokerHand.py** 中添加方法**has\_pair**、**has\_twopair** 等。它们根据手牌是否达到相对应的条件来返回**True**或**False**。你的代码应当对任意数量的手牌都适用（虽然最常见的手牌数是5或7）。

4. 编写一个函数**classify**（分类），它可以弄清楚一副手牌中出现的最大的组合，并设置**label** 属性。例如，一副7张牌的手牌可能包含一个顺子以及一个对子；它应当标记为“**flush**”（顺子）。

5. 当你确保分类方法可用时，下一步是预测各种手牌的概率。在**PokerHand.py** 中编写一个函数，对一副牌进行洗牌，将其分成不同手牌，对手牌进行分类，并记录每种分类出现的次数。

6. 打印一个表格，展示各种分类以及它们的概率。更多次地运行你的程序，直到输出收敛到一个合理程度的正确性为止。将你的结果

和[http://en.wikipedia.org/wiki/ Hand\\_rankings](http://en.wikipedia.org/wiki/Hand_rankings)上的值进行对比。

解答: <http://thinkpython2.com/code/PokerHandSoln.py>。

## 第19章 Python拾珍

本书的一大目标一直是尽可能少地介绍Python语言。如果做某种事情有两种方法，我会选择一种，并避免提及另一种。或者有时候，我会把另一种方法作为练习进行介绍。

本章我会带领大家回顾那些遗漏的地方。Python提供了不少并不是完全必需的功能（不用它们也能写出好代码），但有时候，使用这些功能可以写出更简洁、更可读或者更高效的代码，甚至有时候三者兼得。

### 19.1 条件表达式

我们在5.4节中见过条件语句。条件语句通常用来从两个值中选择一个。例如：

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

这条语句检查`x` 是否为正数。如果为正数，则计算`math.log`；如果为负数，`math.log` 会抛出`ValueError` 异常。为了避免程序停止，我们直接生成一个“NaN”，一个特殊的浮点数，代表“不是数”（Not A Number）。

我们可以用**条件表达式**来更简洁地写出这条语句：

```
y = math.log(x) if x > 0 else float('nan')
```

这条语句几乎可以用英语直接读出来：“y gets log-x if x is greater than 0; otherwise it gets NaN”（Y 的值在x 大于0 时是`math.log(x)`， 否则是NaN）。

递归函数有时候可以用条件表达式重写。例如，下面是 `factorial` 的一个递归版本：

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

我们可以将其重写为：

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

条件表达式的另一个用途是处理可选参数。例如，下面是 `GoodKangaroo` 的 `init` 方法（参见练习17-2）：

```
def __init__(self, name, contents=None):
    self.name = name
    if contents == None:
        contents = []
    self.pouch_contents = contents
```

我们可以将其重写为：

```
def __init__(self, name, contents=None):
    self.name = name
    self.pouch_contents = [] if contents == None else contents
```

一般来说，如果条件语句的两个条件分支都只包含简单的返回或对同一变量进行赋值的表达式，那么这个语句可以转化为条件表达式。

## 19.2 列表理解

在10.7节中我们已经见过映射和过滤模式。例如，下面的函数接收一个字符串列表，将每个元素通过字符串方法`capitalize`进行映射，并返回一个新的字符串列表。：

```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

我们可以用**列表理解**（list comprehension）把这个函数写得更紧凑：

```
def capitalize_all(t):
    return [s.capitalize() for s in t]
```

上面的方括号操作符说明我们要构建一个新列表。方括号之内的表达式指定了列表的元素，而**for**子句则表示我们要遍历的序列。

列表理解的语法有一点粗糙的地方，因为里面的循环变量，即本例中的`s`，在表达式中出现在定义之前。

列表理解也可以用于过滤操作。例如，下面的函数选择列表`t`中的大写元素，并返回一个新列表：

```
def only_upper(t):  
    res = []  
    for s in t:  
        if s.isupper():  
            res.append(s)  
    return res
```

我们可以用列表理解将其重写为：

```
def only_upper(t):  
    return [s for s in t if s.isupper()]
```

对于简单表达式来说，列表理解更紧凑、更易于阅读，并且它们通常都比实现相同功能的循环更快，有时候甚至快很多。因此，如果你因为我没有早些提到它而恼怒，我表示十分理解。

但是我得辩解一下，列表理解更难以调试，因为你没法在循环内添加打印语句。我建议你只在计算简单到一次就能弄对的时候才使用它。对于初学者来说，这意味着从来不用。

## 19.3 生成器表达式

**生成器表达式**（generator expression）和列表理解类似，但是它使用圆括号，而不是方括号：

```
>>> g = (x**2 for x in range(5))
>>> g
<generator object <genexpr> at 0x7f4c45a786c0>
```

结果是一个生成器对象，它知道该如何遍历值的序列。但它又和列表理解不同，它不会一次把结果都计算出来，而是等待请求。内置函数`next` 会从生成器中获取下一个值：

```
>>> next(g)
0
>>> next(g)
1
```

当到达序列的结尾后，`next` 会抛出一个`StopIteration` 异常。可以使用`for` 循环来遍历所有值：

```
>>> for val in g:
...     print(val)
4
9
16
```

生成器对象会跟踪记录访问序列的位置，所以`for` 循环会从上一个`next` 所在的位置继续。一旦生成器遍历结束，再访问它就会抛出`StopException`：

```
>>> next(g)
StopIteration
```

生成器表达式经常和`sum`、`max` 和`min` 之类的函数配合使用：



```
>>> sum(x**2 for x in range(5))
30
```

## 19.4 any 和all

Python提供了一个内置函数`any`，它接收一个由布尔值组成的序列，并在其中任何值是`True`时返回`True`。它可以用于列表：

```
>>> any([False, False, True])
True
```

但它更常用于生成器表达式：

```
>>> any(letter == 't' for letter in 'monty')
True
```

上面这个例子用处不大，因为它做的事情和`in`表达式一样。但是我们可以用`any`来重写9.3节中的搜索函数。例如，我们可以将`avoids`函数重写为：

```
def avoids(word, forbidden):
    return not any(letter in forbidden for letter in word)
```

这个函数读起来几乎和英语一致：“`word` avoids forbidden if there are not any forbidden letters in word”（我们说一个`word`避免被禁止，是指`word`中没有任何被禁的字母）。

Python还提供了另一个内置函数**all**，它在序列中所有元素都是**True**时返回**True**。作为练习，请使用**all** 重写9.3节中的**uses\_all** 函数。

## 19.5 集合

我曾在13.6节中使用字典来寻找在文档中出现但不属于一个单词列表的单词。我写的函数接收一个字典参数**d1**，其中包含文档中所有的单词作为键；以及另一个参数**d2**，包含单词列表。它返回一个字典，包含**d1** 中所有不在**d2** 之中的键：

```
def subtract(d1, d2):
    res = dict()
    for key in d1:
        if key not in d2:
            res[key] = None
    return res
```

在这些字典中，值都是**None**，因为我们从来不用它们。因此，我们实际上浪费了一些存储空间。

Python还提供了另一个内置类型，称为集合（**set**），它表现得和没有值而只使用键集合的字典类似。向一个集合添加元素很快，检查集合成员也很快。集合还提供方法和操作符来进行常见的集合操作。

例如，集合减法可以使用方法**difference** 或者操作符‘-’来实现。因此我们可以将**subtract** 函数重写为：

```
def subtract(d1, d2):  
    return set(d1) - set(d2)
```

结果是一个集合而不是字典，但是对于遍历之类的操作，表现是一样的。

本书中的一些练习可以用集合来更加简洁且高效地实现。例如，练习10-7中的`has_duplicates` 函数，下面是使用字典来实现的一个解答：

```
def has_duplicates(t):  
    d = {}  
    for x in t:  
        if x in d:  
            return True  
        d[x] = True  
    return False
```

一个元素第一次出现的时候，把它加入到字典中。如果相同的元素再次出现时，函数就返回`True`。

使用集合，我们可以这样写同一个函数：

```
def has_duplicates(t):  
    return len(set(t)) < len(t)
```

一个元素在一个集合中只能出现一次，所以如果`t` 中间的某个元素出现超过一次，那么变成集合后其长度会比`t` 小。如果没有任何重复元素，那么集合的长度应当和`t` 相同。

我们也可以使用集合来解决第9章中的一些练习。例如，下面是 `uses_only` 函数使用循环来实现的版本：

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` 检查 `word` 中所有的字符是不是在 `available` 中出现。我们可以这样重写：

```
def uses_only(word, available):
    return set(word) <= set(available)
```

操作符 `<=` 检查一个集合是否是另一个集合的子集，包括两个集合相等的情况。这正好符合 `word` 中所有字符都出现在 `available` 中。

## 19.6 计数器

计数器（`counter`）和集合类似，不同之处在于，如果一个元素出现超过一次，计数器会记录它出现了多少次。如果你熟悉多重集（`multiset`）这个数学概念，就会发现计数器是多重集的一个自然的表达方式。

计数器定义在标准模块 `collections` 中，所以需要导入它再使用。可以用字符串、列表或者其他任何支持迭代访问的类型对象来初始化计数器：

```
>>> from collections import Counter
>>> count = Counter('parrot')
>>> count
Counter({'r':2, 't': 1, 'o': 1, 'p': 1, 'a': 1})
```

计数器有很多地方和字典相似。它们将每个键映射到其出现次数。和字典一样，键必须是可散列的。

但和字典不同的是，在访问计数器中不存在的元素时，它并不会抛出异常。相反，它会返回0：

```
>>> count['d']
0
```

我们可以使用计数器来重写练习10-6中的`is_anagram`函数：

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

如果两个单词互为回文，则它们会包含相同的字母，且各个字母的计数相同，所以它们对应的计数器对象也会相等。

计数器提供方法和操作符来进行类似集合的操作，包括集合加法、减法、并集和交集。计数器还提供一个非常常用的方法`most_common`，它返回一个值-频率对的列表，按照最常见到最少见来排序：

```
>>> count = Counter('parrot')
>>> for val, freq in count.most_common(3):
...     print(val, freq)
r 2
p 1
```

```
a 1
```

## 19.7 defaultdict

`collections` 模块还提供了 `defaultdict`，它和字典相似，不同的是，如果你访问一个不存在的键，它会自动创建一个新值。

创建一个 `defaultdict` 对象时，需要提供一个用于创建新值的函数。用来创建对象的函数有时被称为**工厂**（**factory**）函数。用于创建列表、集合以及其他类型对象的内置函数，都可以用作工厂函数：

```
>>> from collections import defaultdict
>>> d = defaultdict(list)
```

请注意，参数是 `list`（一个类对象），而不是 `list()`（一个新的列表）。你提供的函数直到访问不存在的键时，才会被调用的：

```
>>> t = d['new key']
>>> t
[]
```

新列表 `t` 也会加到字典中。所以，如果我们修改 `t`，改动也会在 `d` 中体现：

```
>>> t.append('new value')
>>> d
defaultdict(<class 'list'>, {'new key': ['new value']})
```

如果创建一个由列表组成的字典，使用`defaultdict` 往往能够帮你写出更简洁的代码。在练习12-2的解答中，我创建了一个字典，将排序的字母字符串映射到可以由那些字母拼写出来的单词列表。例如，`'opst'` 映射到列表`['opts', 'post', 'pots', 'spot', 'stop', 'tops']`。可以从[http://thinkpython2.com/code/anagram\\_sets.py](http://thinkpython2.com/code/anagram_sets.py)下载该解答。

下面是原始的代码：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

这个函数可以用`setdefault` 简化，你可能在练习11-2中也用过：

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d.setdefault(t, []).append(word)
    return d
```

但这个解决方案有一个缺点，它不管是否需要，每次都会新建一个列表。对于列表来说，这并不算大问题，但如果工厂函数非常复杂，就有可能成为问题了。

我们可以使用`defaultdict` 来避免这个问题，并进一步简化代码：

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

在练习18-3的解答中，函数`has_straightflush` 中使用了`setdefault`。可以从[http:// thinkpython2.com/code/PokerHandSoln.py](http://thinkpython2.com/code/PokerHandSoln.py) 下载它。但这个解决方案的缺点是，不管是否必需，每次循环迭代都会创建一个新的`Hand` 对象。作为练习，请使用`defaultdict` 重写该函数。

## 19.8 命名元组

很多简单的对象其实都可以看作是几个相关值的集合。例如，第15章中定义的`Point` 对象，包含两个数字，即`x` 和`y`。定义一个这样的类时，通常会从`init` 方法和`str` 方法开始：

```
class Point:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __str__(self,):
        return '(%g, %g)' % (self.x, self.y)
```



这里用了很多代码来传达很少的信息。Python提供了一个更简洁的方式来表达同一个意思：

```
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
```

第一个参数是你想要创建的类名。第二个参数是Point对象应当包含的属性的列表，以字符串表示。namedtuple 的返回值是一个类对象：

```
>>> Point
<class '__main__.Point'>
```

这里Point类会自动提供\_\_init\_\_ 和\_\_str\_\_ 这样的方法，所以你不需写它们。

要创建一个Point对象，可以把Point类当作函数来用：

```
>>> p = Point(1, 2)
>>> p
Point(x=1, y=2)
```

init 方法使用你提供的名字把实参值赋给属性。str 方法会打印出Point对象及其属性的字符串表示。

可以使用名称来访问命名元组的元素：

```
>>> p.x, p.y
(1, 2)
```

也可以直接把它当作元组来处理：

```
>>> p[0], p[1]
(1, 2)

>>> x, y = p
>>> x, y
(1, 2)
```

命名元组提供了快速定义简单类的方法，但其缺点是简单的类并不会总保持简单。可能之后你需要给命名元组添加方法。如果那样，可以定义一个新类，继承当前的命名元组：

```
class Pointier(Point):
    # 在这里添加更多的方法
```

或者也可以直接切换成传统的类定义。

## 19.9 收集关键词参数

在12.4节中，我们见过如何编写函数将其参数收集成一个元组：

```
def printall(*args):
    print(args)
```

可以使用任意个数的按位实参（也就是说，不带名称的实参）来调用这个函数：

```
>>> printall(1, 2.0, '3')
(1, 2.0, '3')
```

但是\*号操作符并不会收集关键词实参：

```
>>> printall(1, 2.0, third='3')
TypeError: printall() got an unexpected keyword argument 'third'
```

要收集关键词实参，可以使用\*\*操作符：

```
def printall(*args, **kwargs):
    print(args, kwargs)
```

这里收集关键词形参可以任意命名，但`kwargs` 是一个常见的选择。收集的结果是一个将关键词映射到值的字典：

```
>>> printall(1, 2.0, third='3')
(1, 2.0){'third': '3'}
```

如果有一个关键词到值的字典，就可以使用分散操作符\*\* 来调用函数：

```
>>> d = dict(x=1, y=2)
>>> Point(**d)
Point(x=1, y=2)
```

没有用分散操作符的话，函数会把`d` 当作一个单独的按位实参，所以它会把`d` 赋值给`x`，并因为没有提供`y` 的赋值而报错：

```
>>> d = dict(x=1, y=2)
>>> Point(d)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __new__() missing 1 required positional argument: 'y'
```

当处理参数很多的函数时，创建和传递字典来指定常用的选项是非常有用的。

## 19.10 术语表

条件表达式（**conditional expression**）：一个根据条件返回一个或两个值的表达式。

列表理解（**list comprehension**）：一个以方框包含一个**for** 循环，生成新列表的表达式。

生成器表达式（**generator expression**）：一个以括号包含一个**for** 循环，返回一个生成器对象的表达式。

多重集（**multiset**）：一个用来表达从一个集合的元素到它们出现次数的映射的数学概念。

工厂函数（**factory**）：一个用来创建对象，并常常当作参数使用的函数。

## 19.11 练习

### 练习19-1

下面的函数可以递归地计算二项式系数：

```
def binomial_coeff(n, k):  
    """计算(n, k)的二项式系数。
```

```
n: 试验次数
k: 成功次数

返回: int
"""
if k == 0:
    return 1
if n == 0:
    return 0

res = binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
return res
```

使用内嵌条件表达式来重写该函数。

注意：这个函数效率不高，因为它会不停地重复计算相同的值。可以通过使用备忘（**memoizing**，参见11.6节）来提高它的效率。但你可能会发现，使用条件表达式之后，添加备忘会变得比较困难。

## 第20章 调试

调试程序时，应当区分不同类型的错误，以便更快地查找出错误原因。

- 语法错误（**semantic error**）在将源代码翻译为字节码的过程中由解释器发现。它们通常表示有程序结构错误。例如，在**def** 语句的末尾漏掉冒号，会产生一个有些冗余的错误信息  
**SyntaxError: invalid syntax**。
- 运行时错误（**runtime error**）由解释器在程序运行的过程中发现错误后产生。大部分错误消息都包含了错误发生的位置以及正在执行的函数的信息。例如：一个无限递归最终会导致运行时错误  
**maximum recursion depth exceeded**（超过最大递归深度）。
- 语义错误（**semantic error**）是程序运行中没有产生错误信息，但做的事情却不正确的情况。例如：一个表达式求值的顺序和你预想的不同，因此产生了不正确的结果。

调试的第一步就是弄清楚你面对的到底是哪种类型的错误。虽然下面的几节是按照错误类型来组织的，但有些技巧其实可以适用于多种情形。

### 20.1 语法错误

语法错误，在弄清楚它们是什么之后，通常都很容易修正。不幸的是，错误信息往往没什么帮助。最常见的错误信息是 `SyntaxError: invalid syntax` 和 `SyntaxError: invalid token`，这两种都没多少信息量。

另一方面，信息也确实告诉你问题在程序中发生的位置。实际上，它告诉你的是 `Python` 发现错误的位置，而并不一定总和错误发生的位置相同。有时候错误发生在错误信息指明的位置之前，往往是前一行。

如果你递增地构建程序，应当很清楚错误发生的位置。它常常在你最后添加的那行代码上。

如果你是从书本中复制代码，则最好先仔细比较自己的代码和书中的代码。检查每一个字母。同时请记得书本也可能是错的，所以如果你看到一个像是语法错误的东西，那么它有可能就是。

下面是一些可以避免最常见的语法错误的方法。

1. 确保你没有使用 `Python` 关键字作为变量名称。
2. 检查在每一个复合语句的语句头结尾，都有一个冒号，包括 `for`、`while`、`if` 和 `def` 语句。
3. 确保程序中每个字符串都有前后匹配的引号。确定每个括号都是直引号（如`"`），而不是弯引号（如`"`）。

4. 如果有三引号（单引号或双引号字符）多行字符串，确保你正确结束了字符串。没有正确结束的字符串，会导致程序结尾处产生 `invalid token` 错误，或者它会将接下来的程序看作字符串的一部分，直到遇到下一个字符串为止。这种情况下，可能都不会产生错误信息！

5. 没有关闭的开始符号（`(`、`{` 或 `[`）会让Python继续解析下一行，并当作当前语句的一部分。通常来说，会在下一行立即产生一个错误。

6. 检查在条件判断时将 `'=='` 写成 `'=` 的经典错误。

7. 检查缩进，确保它们是按照设想正确排布的。Python可以处理空格和制表符，但如果混合使用它们，则可能产生问题。避免这种问题最好的办法是使用一个懂得Python的编辑器，并由它产生一致的缩进。

8. 如果你的代码中有非ASCII字符（包括字符串和注释中），虽然Python 3通常能处理好非ASCII字符，但还是可能导致问题。当你从网页或其他来源直接复制文本时，需要格外注意。

如果上面的办法都没用，请继续看下一节。

## 我一直进行修改，但没有什么区别

如果解释器报出一个错误而你又找不到，有可能是因为解释器和你用的并不是同一套代码。检查你的编程环境，确保你正在编辑的代码和Python运行的是同一个。



如果不确定，可以尝试在程序开头加上一个明显而故意的错误。再运行一次。如果解释器并没有发现新的错误，那么说明你运行的不是新代码。

可能有以下几种原因。

- 你编辑了代码，但忘了保存更改就直接运行了。有的编程环境会帮你自动保存，有的不会。
- 你修改了文件名，但仍然在使用旧文件名运行程序。
- 你的编程环境可能没有正确配置。
- 如果你在编写一个模块，并使用 `import`，请确保你的模块名称没有和Python标准模块冲突。
- 如果你在使用 `import` 来读入模块，请记得重载一个修改过的文件时，需要重启解释器或者使用 `reload`。如果你直接重新导入这个模块，它并不会做任何事。

如果你遇到困难被卡住，而且弄不清楚到底怎么回事，一个办法是重新以最简单的类似“Hello, World!”的程序开始，并确保你能让一个已知的程序正确运行。然后逐渐添加原先程序的部分到新的程序中。

## 20.2 运行时错误

一旦你的程序已经确保语法正确，Python可以读它，并且至少可以开始运行它。这时候可能发生哪些错误？

### 20.2.1 我的程序什么都不做

这个问题最常见的原因是你的文件包含了各种函数和类的定义，但没有实际调用函数来启动执行。如果你是为了导入模块使用它们提供的类和函数，那么这么做可能是故意的。

如果不是故意的，则确保在程序中有一个函数调用，并确保执行流程能到达这一函数调用（参见20.2.5节）。

## 20.2.2 我的程序卡死了

如果一个程序突然停止并看起来什么事情都没做，它就“卡死了”。通常这意味着程序掉入一个死循环或者无限递归中。

- 如果怀疑一个特别的循环可能是问题所在，可以在循环开始前添加一个**print** 语句，打出“进入循环”，在循环结尾处之后也添加一个，打出“退出循环”。

再次运行程序。如果你看到第一个输出，而没有看到第二个，说明你确实遇到一个死循环了。无限循环的内容参见20.2.3节。

- 大部分情况下，无限递归都会让程序运行一会儿，然后产生“**RuntimeError: Maximum recursion depth exceeded**”错误。如果发生这种情况，参见20.2.4节。

如果你没有看到这个错误，但怀疑可能是递归方法或函数产生的问题，也同样可以使用20.2.4节中的技巧。

- 如果上面两步都没用，尝试其他循环或其他递归方法与函数。

- 如果这些都没用，说明可能是你没理解你的程序的执行流程。执行流程的内容参见20.2.5节。

### 20.2.3 无限循环

如果你觉得有一个无限循环并知道是哪个循环导致的问题，可以在循环的结尾处添加一个**print** 语句，打印出循环条件中的变量值，以及条件的值。

例如：

```
while x > 0 and y < 0 :  
    # do something to x  
    # do something to y  
  
    print('x: ', x)  
    print('y: ', y)  
    print("condition: ", (x > 0 and y < 0))
```

现在当你再次运行程序时，能够看到每次循环中打印出的3行输出。最后一次循环时，条件应该变为**False**。如果循环一直进行，你应当可以看到x 和y 的值，并可能弄清楚为什么它们没有被正确更新。

### 20.2.4 无限递归

大部分情况下，无限递归会导致程序运行一会儿，然后产生**Maximum recursion depth exceeded** 的错误。

如果你怀疑一个函数导致了无限递归，保证递归确实有一个基准情形。应该有一个条件能导致函数直接返回而不再继续递归调用。如

果没有，那么你可能需要重新思考算法，并定位一个基准情形。

如果有一个基准情形，但程序似乎没有到达它，可以在函数的开头加一个**print** 语句来打印参数。现在当你重新运行程序时，会看到每次函数调用时都会打出几行输出，并能看到每次调用的参数值。如果参数并没有向基准情形变化，你大概能发现为何如此。

### 20.2.5 执行流程

如果你不确认程序中的执行流程如何走向，可以在每个函数的开头添加一个**print** 语句，打印类似“进入函数**foo**”之类的输出。这里**foo** 是函数名。

现在如果你重新运行程序，它会打印出每个函数调用的轨迹。

### 20.2.6 当我运行程序，会得到一个异常

如果在运行时遇到一个问题，**Python**会打印出一个信息，包含错误的名称，程序中发生这个错误的位置，以及一个回溯。

回溯里标明了当前执行的函数，以及调用它的函数，以及调用这个调用者的函数，依此类推。换句话说，它回溯了从程序开头直到错误发生所在位置的整个调用轨迹，包括了每个函数所在文件中的行号。

第一步是检查程序中错误发生的位置，并尝试弄清楚问题所在。下面是一些常见的运行时错误。

**NameError**

你在试图使用一个当前环境中并不存在的变量。检查变量名是否有拼写正确，或至少是一致的。请记得局部变量是局部的，不能在定义它们的函数之外使用。

## **TypeError**

有3种可能的原因。

- 你在尝试错误地使用一个值。例如，使用不是整数的值来索引字符串、列表或元组。
- 格式字符串中，内部的格式项和传入的参数不匹配。当格式项的数目不对或者转换的类型不对时都可能发生。
- 调用函数时使用了错误数量的参数。对于方法来说，查看方法定义并检查第一个参数是否为**self**。接着查看方法调用；确保你是在正确类型的对象上调用方法，并正确提供了其他参数。

## **KeyError**

你在试图用一个字典并不包含的键来查找字典的元素。如果键是字符串，请注意大小写问题。

## **AttributeError**

你在尝试访问一个并不存在的属性或方法。检查拼写！你可以使用内置的**vars** 函数来列出存在的属性。

如果**AttributeError**指明一个对象是**NoneType**，则意味着它是**None**。那么问题不是属性名而是对象。

对象为`None`的原因可能是你忘了从函数里返回值；如果函数执行到结尾都没有遇到`return` 语句，那么它会返回`None` 。另一个常见的原因是使用了一个返回`None` 的列表方法作为结果，如`sort` 。

## IndexError

你在访问列表、字符串或元组时使用的索引大于它的长度减一。在错误发生的前一行，添加一个`print` 语句展示索引的值和数组的长度。数组长度是否正确？索引大小是否正确？

Python调试器（`pdb`）在查找异常时很有用，因为它让你可以在错误发生之前的地方查看程序的状态。可以在<http://docs.python.org/3/library/pdb.html>阅读`pdb` 的相关资料。

## 20.2.7 我添加了太多`print` 语句，被输出淹没了

使用`print` 语句进行调试的问题之一是你可能被太多的输出所埋没。有两种方法可以继续：简化输出，或者简化程序。

要简化输出，可以删除或注释掉没用的`print` 语句，或者将它们合并起来，或者格式化输出让它们更容易看懂。

要简化程序，有几件事情可做。首先，简化程序所处理的问题。例如，如果你在搜索一个列表，就改为搜索一个很小的 列表。如果程序从用户获得输入，则输入可以产生错误的最简单的输入。

其次，清理程序。删除无效代码，并重新组织代码让它尽可能更可读。例如，如果你怀疑问题出在程序的一个很深的嵌套部分中，则

应当尝试重写那部分，让它的结构更简单。如果你怀疑一个很大的函数，则尝试将它拆分为多个更小的函数，并分别测试它们。

找寻最简测试用例的过程往往能带你找到问题所在。如果发现程序在一种情况下正常工作，而在另一种情况下则不能，那这些情况本身就给你一些线索。

类似地，重写一部分代码可以帮你找到细微的bug。如果你做出一个认为不该影响程序的改变，而它确实出问题了，这就给了具体的提示。

## 20.3 语义错误

从某种角度看，语义错误更难调试，因为解释器并不提供任何信息。只有你自己知道程序到底应该怎么做。

解决语义错误的第一步是在程序文本和你看到的程序行为之间建立一个连接。你需要对程序实际在做什么有一个假设。让这件事情很难的原因之一是计算机运行得太快。

你常常会希望程序能够减慢到人的速度，而使用调试器时你可以做到。但往程序里插入几条精确放置的`print` 语句，比起设置调试器，插入或删除断点，并“单步”执行到程序出错的地方，往往花费的时间更少。

### 20.3.1 我的程序运行不正确

你应该问自己如下几个问题。

- 程序中有没有地方你期望它去做而实际上没有发生的？找到运行那段功能的代码，并确保它确实如你所期望的那样运行了。
- 有没有一些不应该发生的事情？找到程序中运行了某种不该出现的功能的代码。
- 有没有一段代码产生的效果和你所期望的不一致？确保你完全明白该段代码，特别是当它牵涉到其他Python模块的函数或方法时。阅读你调用的函数的文档。使用简单的测试用例测试它们并检查结果。

为了能够编程，你需要程序如何工作的一个思维模型。如果编写出一段和你预期不同的代码，常常问题不是在程序本身，而是在你的思维模型上。

修正你的思维模型的最佳方法是将程序划分成不同部分（通常是函数和方法）并独立测试每一个部分。一旦找到你的模型和真实世界的偏差，就能够解决问题了。

当然，在开发程序时你应当分组件进行构建和测试。如果发现一个问题，应该只需要检查一小部新的不确认是否正确的代码。

### 20.3.2 我有一个巨大而复杂的表达式，而它和我预料的不同

编写复杂的表达式并没有问题，只要能保证它们还可读。但它们也会变得更难调试。将复杂的表达式拆分成一系列的赋值到临时变量的语句，常常是个好主意。

例如：



```
self.hands[i].addCard(self.hands[self.findNeighbor(i)].popCard())
```

这个表达式可以写作：

```
neighbor = self.findNeighbor(i)
pickedCard = self.hands[neighbor].popCard()
self.hands[i].addCard(pickedCard)
```

后面更清晰的版本也更加可读，因为变量名称提供了附加的文档信息，它也更容易调试，因为你可以检查中间变量的类型，并打印它们的值。

复杂表达式的另一个问题是求值的顺序可能和你所期望的不同。例如，如果你将表达式 $x/2\pi$ 翻译成Python，可能会这么写：

```
y = x / 2 * math.pi
```

这样并不正确，因为乘法和除法有相同的优先级，并且语句求值的顺序是从左至右。所以这个表达式计算的实际上是 $x\pi/2$ 。

调试表达式的一个好办法是添加括号来显式控制求值顺序：

```
y = x / (2 * math.pi)
```

任何时候如果不确定求值的顺序，都可以使用括号。这样不但会让程序更加正确（从按照你的设想来做的角度说），也会让其他人更容易阅读你的代码，因为不需要去记忆操作的顺序。

### 20.3.3 我有一个函数，返回值和预期不同

如果你在程序中有`return`语句返回一个复杂的表达式，则没有机会在返回之前打印结果。这时候，也可以使用临时变量。例如，这个语句：

```
return self.hands[i].removeMatches()
```

可以写作：

```
count = self.hands[i].removeMatches()  
return count
```

现在你有机会在返回之前显示`count`的值了。

### 20.3.4 我真的真的卡住了，我需要帮助

首先，试着离开计算机几分钟。计算机会发射辐射影响大脑，产生下列症状。

- 挫败感和愤怒感。
- 迷信的信念（“我的计算机恨我”）和神奇的想法（“程序只有在我反戴帽子时才正确运行”）。
- 随机行走编程（尝试着写下所有可能的程序，并选择运行正确的那个）。

如果你发现自己正在遭受这些症状之一，请马上站起来出去散个步。当你平静下来后，再思考程序。它在做什么？产生那种行为的可能原因有哪些？上一次程序还正确运行是什么时候，之后你做了什么？

有时候发现一个bug确实需要时间。我常常能够在远离计算机并让思维休息之后找到bug。找到错误的最佳地点有火车上、浴缸中及将要入睡之前在床上。

### 20.3.5 不行，我真的需要帮助

这种事确实会发生。即使最好的程序员也会偶尔卡住。有时候你在一段程序上工作太久了所以反而看不到错误。你需要一双新的眼睛。

在叫人帮忙之前，请确保你已经准备好。你的程序应当尽量简单，而你应当使用最小的输入来复现错误。你应当在合适的地方放好了`print` 语句（并且它们的输出应当容易理解）。你应当足够理解这个问题，因此能够简明扼要地描述它。

当你找人帮忙时，请确保给他们需要的信息。

- 如果有错误信息，它是什么，它代表了程序的哪部分？
- 在这个错误发生之前，你做的最后一件事情是什么？你写的最后一段代码是什么？失败的新测试用例是什么？
- 目前为止你做了哪些尝试，并从中得到了什么？

当你找寻bug时，思考一下如何做才能找得更快。下一次见到类似的情形时，就能够更快地找到问题了。

记住，目标不只是让程序正确运行。目标是学会如何让程序正确运行。



## 第21章 算法分析

这个附录编选自O'Reilly Media出版的Allen B. Downey的*Think Complexity*（2012）一书。当你读完本书之后，可能会想继续读那本书。

**算法分析** 是计算机科学的一个分支，研究算法的性能，尤其是它们的运行时间和空间需求。参见 [http://en.wikipedia.org/wiki/Analysis\\_of\\_algorithms](http://en.wikipedia.org/wiki/Analysis_of_algorithms)。

算法分析的实践目标是预测不同算法的性能，以便于指导设计决策。

在2008年的美国总统大选中，候选人巴拉克·奥巴马在访问Google公司时被要求做一个即兴分析。Google的首席执行官埃里克·施密特问他“给100万个32位整数排序的最高效算法”是什么。奥巴马显然被提示了，因为他马上回答，“我觉得冒泡排序可能是错误的做法”。参见 <http://bit.ly/1MpIwTf>。

这是真的：冒泡排序在概念上很简单，但对于大数据量的排序很慢。施密特想得到的答案可能是“基数排序”（[http://en.wikipedia.org/wiki/Radix\\_sort](http://en.wikipedia.org/wiki/Radix_sort)）<sup>[1]</sup>。

算法分析的目标是在不同算法间做出有意义的比较，但也有一些问题。

- 算法的相对性能可能依赖于硬件的特征，所以一个算法可能在机器A上更快，另一个在机器B上更快。这个问题的通用解决方法是先指定一个**机器模型**，并分析在一个指定的机器模型中一个算法需要执行的步骤或操作。
- 相对性能还可能依赖于数据集的细节特征。例如，有的排序算法在数据已经是部分排序的情形下比其他算法更快，有的程序在这种情况下反而慢。避免这个问题的通常办法是分析**最坏情况**场景。有时候分析平均情况的性能也有用，但也通常会更难，因为有哪些情形可以用来“平均”往往并不明显。
- 相对性能也依赖于问题的规模。对小序列更快的排序算法可能对大序列就慢了。这个问题的通常解决方案是用一个问题规模的函数来表达运行时间（或操作数），并根据问题规模增大的速度将函数进行归类。

这种比较的好处之一是自然而然地可以将算法进行简单地分类。例如，如果我知道算法A的运行时间趋向于和输入的规模 $n$ 成比例，而算法B趋向于和 $n^2$ 成比例，那么我会预期至少对于大的 $n$ 值，算法A比算法B快。

这种分析也有需要注意的地方，后面会谈到。

## 21.1 增长量级

假设你需要分析两个算法，并依照输入的规模来表达它们的运行时间：算法A需要 $100n + 1$ 步来解决规模为 $n$ 的问题，算法B需要 $n^2 + n + 1$ 步。

下面的表格显示了这两个算法在不同的问题规模下的运行时间：

输入规模	算法A的运行时间	算法B的运行时间
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$>10^{10}$

在 $n=10$ 时，算法A看起来很差；它几乎需要10倍于算法B的时间。但对于 $n=100$ 来说它们就已经差不多了，而在更大的规模时，算法A远好于算法B。

这里根本的原因在于对很大的 $n$ 值，任何包含 $n^2$ 项的函数都会比首项是 $n$ 的函数增长快速很多。**首项**是一个多项式中最高次方的项。

对于算法A，首项有一个很大的系数100，因此算法B在小的 $n$ 时比算法A快。但不论系数是多少，总有一个 $n$ 值会导致 $an^2 > bn$ 。

对于非首项来说也如此。即使算法A的运行时间是 $n+1000000$ ，对于足够大的 $n$ ，仍然会比算法B快。

总的来说，我们预期有更小的首项的算法对大规模问题来说是更好的算法。但对于小一些的问题来说，可能存在一个**交叉点**，那里其

他算法可能更好。交叉点的位置取决于算法的细节、输入以及硬件的条件，所以在算法分析时常常被忽略掉。但那并不意味着你可以忘记它。

如果两个算法有相同的首项，则很难说哪一个更好；同样地，答案也取决于细节条件。所以对于算法分析来说，首项相同的函数被认为是同等的，即使它们的系数不同。

**增长量级** 就是各种增长行为被认为是同等的函数的集合。例如， $2n$ 、 $100n$  和  $n+1$  都是一个增长量级，用**大O标记法** 写作  $O(n)$ ，通常称为**线性的**，因为这个集合中的每个函数都依据  $n$  线性增长。

所有首项是  $n^2$  的函数都属于  $O(n^2)$ ，它们被称为是**平方的**。

下面的表格显示了算法分析中大部分最常见的增长量级，按照更坏的程度递增：

增长量级	名称
$O(1)$	常量级
$O(\log_b n)$	对数级（对任意 $b$ ）
$O(n)$	线性级
$O(n \log_b n)$	$n \log n$



增长量级	名称
$O(n^2)$	平方级
$O(n^3)$	立方级
$O(c^n)$	指数级（底数 $c$ 任意）

对于对数项，底数并没有影响；修改底数相当于乘以一个常量，而那样并不影响增长量级。类似地，所有的指数函数都是同一个增长量级，不论指数的底数是什么。指数函数增长非常迅速，所以指数级算法只在小规模问题中应用。

### 练习21-1

在[http://en.wikipedia.org/wiki/Big\\_O\\_notation](http://en.wikipedia.org/wiki/Big_O_notation)上阅读大O标记法的维基百科页面，并回答下列问题。

1.  $n^3 + n^2$  的增长量级是多少？ $1000000n^3 + n^2$  呢？ $n^3 + 1000000n^2$  呢？
2.  $(n^2 + n) \cdot (n + 1)$  的增长量级是多少？在相乘之前，请记住你只需要首项。
3. 如果 $f$ 是 $O(g)$ ，对于未指定的函数 $g$ ，我们怎么说 $af + b$ ？
4. 如果 $f_1$ 和 $f_2$ 都是 $O(g)$ ，那么 $f_1 + f_2$ 呢？

5. 如果 $f_1$  是 $O(g)$ 而 $f_2$  是 $O(h)$ , 那么 $f_1 + f_2$  呢?

6. 如果 $f_1$  是 $O(g)$ 而 $f_2$  是 $O(h)$ , 那么 $f_1 \cdot f_2$  呢?

关心程序性能的程序员常常会觉得这种分析很难理解。他们有道理：有时候系数和非首项也能带来不同。有时候硬件的细节、编程语言，以及输入的特征，都能带来很大的区别。并且对于小规模问题来说，渐进行为是无关要紧的。

但如果在脑中记着这些需要注意的要点的话，算法分析毕竟是一个有用的工具。至少对于大规模问题来说，“更好”的算法往往确实更好，并且有时候它会好得多。两个增长量级相同的算法的区别往往是一个常量值，但一个好算法和一个坏算法的差距是没有界限的！

## 21.2 Python基本操作的分析

在Python中，大部分算术操作都是常量时间的；乘法通常比加法和减法花费更多时间，而除法花费的更多，但这些操作的时间与参数的大小无关。特别大的整数是一个例外，在那种情况下，运行时间随着数字的位数增加而增加。

索引操作——在序列或字典中读写元素——也是常量时间的，与数据结构的规模无关。

遍历一个序列或字典的for 循环通常是线性的，只要循环体内的操作本身是常量级。例如，将一个列表的元素相加是线性的：

```
total = 0
for x in t:
```

```
total += x
```

内置函数`sum`也是线性的，因为它做相同的事情。但它趋向于更快些，因为实现得更高效；用算法分析的语言来说，就是它有一个更小的首项系数。

作为一个经验规则，如果循环体的增长量级是 $O(n^a)$ 则整个循环是 $O(n^{a+1})$ 。例外情况是当你能够证明循环在一个常量数的迭代之后就能退出。如果不论 $n$ 是多少，循环只最多运行 $k$ 次，则即使对很大的 $k$ 来说，整个循环的增长量级还是 $O(n^a)$ 。

乘以 $k$ 并不会改变增长量级，而除法也不会。所以，如果一个循环体的增长量级是 $O(n^a)$ ，那么它运行 $n/k$ 次，即使对很大的 $k$ 来说，整个循环的增长量级也仍然是 $O(n^{a+1})$ 。

大部分字符串和元组操作都是线性的，只有下标访问和`len`函数例外，它们是常量级时间的。内置函数`min`和`max`是线性的。切片操作的运行时间与输出的长度成正比，而与输入的长度无关。

字符串拼接是线性的，它的运行时间与操作数的长度的总和有关。

所有的字符串方法都是线性的，但如果字符串的长度受限于一个常量（例如，在只有一个字符的字符串的操作），可以看作是常量的。字符串方法`join`是线性的，它的运行时间与字符串的总长度有关。

大多数列表方法是线性的，但也有一些例外。

- 在列表结尾处添加一个元素的操作平均来说是常量时间的；当它空间不足时，偶尔会复制到另一个更大的地方，但总的 $n$ 次操作的时间量级是 $O(n)$ ，所以每次操作的平均时间是 $O(1)$ 。
- 从列表结尾删除一个元素的操作是常量时间的。
- 排序的量级是 $O(n \log n)$ 。

大部分字典操作和方法都是常量时间的，但也有一些例外。

- **update** 的运行时间和作为参数传入的字典的大小成比例，而不是被更新的字典本身。
- **keys**、**values** 和 **items** 都是常量时间，因为它们返回的是迭代器。但是，如果循环遍历这个迭代器，则循环是线性的。

字典的效率是计算机科学的一个小奇迹。我们会在21.4节中介绍它是如何工作的。

## 练习21-2

在[http://en.wikipedia.org/wiki/Sorting\\_algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm)阅读排序算法的维基百科页面并回答下列问题。

1. 什么是“比较排序”？比较排序的最坏情况的增长量级最好是什么？任何排序算法中，最坏情况的增长量级最好是多少？
2. 冒泡排序的增长量级是多少？为什么奥巴马认为它是“错误的做法”？

3. 基数排序的增长量级是多少？要使用它，我们需要哪些前置条件？
4. 稳定排序是什么，为什么在实践中它很重要？
5. 最差的（有名字的）排序算法是什么？
6. C语言库里用的排序算法是什么？Python里用的是什麼？这些算法稳定吗？你可能需要去Google搜索这些答案。
7. 很多非比较排序都是线性的，那么为什么Python会使用 $O(n \log n)$ 的比较排序呢？

## 21.3 搜索算法的分析

**搜索** 是一种算法，接收一个集合和一个目标元素，并决定这个元素是否在集合中，通常返回元素的索引。

最简单的搜索算法是“线性搜索”，即按顺序遍历集合的每一个元素，直到找到目标元素为止。在最坏的情况下，它会遍历整个集合，所以运行时间是线性的。

序列的`in` 操作符使用一个线性搜索；字符串方法`find` 和`count` 也是这样。

如果序列中的元素是排好序的，可以使用**二分查找**，它的增长量级是 $O(\log n)$ 。二分查找和在字典（真实的字典，而不是那个数据结构）中查找单词的算法类似。不像普通搜索那样从第一个元素开始，

它是从序列的中间开始，检查要查找的词是在中间的元素之前还是之后。如果在之前，则继续查找序列的前半段，否则查找后半段。不论哪种情况，都可以将查找的数量减少一半。

如果序列有1 000 000个元素，大概需要花20个步骤找到单词或者发现它不存在。所以那样会比线性查找快大概50 000倍。

二分查找可以比线性查找快很多，但需要序列本身是排好序的，也就需要一些额外工作。

有另一个数据结构，称为**散列表**（hashtable），它甚至更快——它可以用常量时间来搜索——而且不需要元素是排好序的。Python字典是使用散列表实现的，因此大部分字典操作，包括**in** 操作符，都是常量时间的。

## 21.4 散列表

为了解释散列表的工作机制以及为何它的效率如此好，我们先从一个简单的映射实现开始，并逐步改善它，直到成为一个散列表。

我使用Python来展示这些实现。但真实世界中，你不需要用Python写这样的代码，你只需要直接使用字典即可！所以本章中剩下的部分，你需要想象字典并不存在，而你需要实现一个数据结构将键映射到值。你需要实现的操作有以下几个。

**add(k, v)**

添加一个新项，将键 $k$ 映射到值 $v$ 。在Python字典 $d$ 中，这个操作写作 $d[k] = v$ 。

`get(k)`

根据键 $k$ 查找对应的值。在Python字典 $d$ 中，这个操作写作 $d[k]$ 或 $d.get(k)$ 。

就现在来说，我假设每个键只出现一次。最简单的实现是使用一个元组列表，每个元组是一个键值对：

```
class LinearMap:
    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

**add** 往元组列表中添加一项，这个操作是常量时间的。

**get** 使用一个**for** 循环来搜索列表：如果找到了目标键，则返回对应的值；否则抛出**KeyError**。所以**get** 是线性的。

另一个方案是让列表按照键来排序。这样**get** 就可以使用二分查找，其增长量级是 $O(\log n)$ 。但插入一个新项到列表中间是线性的，所

以这可能也不是最好的选择。也有数据结构可以用对数时间实现**add**和**get**，但那仍然没有常量时间好，所以我们继续。

改善**LinearMap**的方法之一是将键值对的列表拆分成更小的列表。下面是一个称为**BetterMap**的实现，它是一个包含100个**LinearMap**的列表。我们接下来会看到，**get**的增长量级仍然是线性的，但是**BetterMap**离散列表更近了一步。

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

**\_\_init\_\_**创建由n个**LinearMap**组成的列表。

**find\_map**被**add**和**get**调用，用来确定用哪个映射来保存新项，或者到哪个映射里去搜索。

**find\_map**使用了内置函数**hash**，它接收几乎所有的Python对象，并返回一个整数。这个实现的限制之一是它只对可散列的键类型可用。可变类型，如列表和字典，是不可散列的。



两个认为相等的可散列对象会返回相同的散列值，但反过来并不一定是真：两个具有不同值的对象可以返回相同的散列值。

`find_map` 使用求余操作符来将散列值封装到0到 `len(self.maps)` 的范围中，这样结果是列表的一个合法索引。当然，这意味着很多不同的散列值会封装到同一个索引上。但如散列函数将对象分配地很均匀（这也是散列函数设计的目标），那么我们预计每个LinearMap有 $n/100$ 个项。

因为LinearMap.get 的运行时间是和其包含的项数成比例的，所以我们预计BetterMap会比LinearMap快100倍。增长量级仍然是线性，但首项系数更小。这很好，但仍然不如散列表好。

下面（终于）是让散列表能变快的关键原因：如果你能保证LinearMap的长度有限，LinearMap.get 则会是常量时间。你需要做的只是记录元素的总数，并当每个LinearMap的大小超过一个阈值时，重新划分散列表，添加更多的LinearMap。

下面是一个散列表的实现：

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
```

```
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

每个**HashMap** 都包含一个**BetterMap**; `__init__`从2个**LinearMap**开始, 并初始化**num**, 它会用来记录总的项数。

**get** 只需要分配到对应的**BetterMap**。真正的工作都发生在**add**中, 它会检查项数和**BetterMap**的大小: 如果相等, 那么每个**LinearMap**的平均项数是1, 所以它调用**resize**。

**resize** 创建一个新的**BetterMap**, 比之前大一倍, 并将旧有的映射中的项“重新散列”到新的映射中。

重新散列是有必要的, 因为**LinearMap**的数量的改变, 导致**find\_map**的求余操作符的分母改变。也就是说, 有些原先会散列到同一个**LinearMap**的项会分配到不同的**LinearMap**中 (这也是我们想要的, 对吧?)。

重新散列是线性的, 所以**resize**是线性的, 看起来可能不好, 因为我保证过**add**应当是常量时间的。但请记住我们并不是每次都需要进行**resize**, 所以**add**通常是常量时间的, 只是偶尔会线性。**add**运行 $n$ 次的总时间是和 $n$ 成比例的, 因此每次调用**add**的平均时间是常量时间!

要明白散列表如何工作，考虑从一个空的**HashTable**开始，并添加一些项。我们从2个**LinearMap**开始，所以最开始两个**add** 会很快（不需要**resize**）。我们说它们每次花费一单位的工作量。下一个**add** 会需要**resize**，所以我们需要重新散列前两项（我们说这需要再加2个单位的工作量）并添加一个新项（再加1个单位）。再添加一项花费1单位，所以至今为止是4项花费了6个单位的工作。

下一个**add** 需要5个单位，但接着的3个都只需要1个单位，所以总共是8项花费了14单位。

再下一个**add** 需要9个单位，但接着我们可以在再次**resize** 之前添加7项，所以总共是16个**add** 花费了30单位。

在32个**add** 时，总共的花费是62单位，而我希望你已经开始看到其中的模式了。在 $n$  个**add** 之后，假设 $n$  是2的乘方，总的花费是 $2n - 2$  单位，所以平均每个**add** 的工作量是稍微小于2个单位的。当 $n$  是2的乘方时，这是最好情况；对于其他的 $n$  值，平均工作量稍高一点，但这并不重要。重要的是这是 $O(1)$ 。

图21-1用图形化的方式展示了这个过程。每个方块代表一个单位的工作量。每一列显示每个**add** 的工作量：从左到右，前两个**add** 花费1单位，第三个花费3单位，等等。

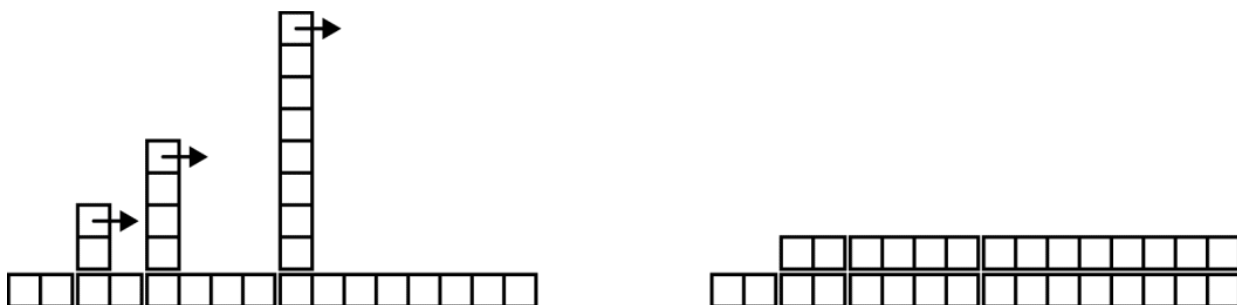


图21-1 散列表add的消耗

多余的重新散列的工作看起来像一序列不断增高的塔，之间的间隔越来越远。现在如果你将塔推倒，将**resize**的花费均摊到所有**add**操作上，就会发现 $n$ 个**add**之后总的花费是 $2n - 2$ 。

这个算法的一个重要特点是当我们调整**HashTable**的大小时，它会几何增长；也就是，我们乘以一个常量到大小上。如果算术地增加大小——每次添加固定数量的数——那么每个**add**的平均时间是线性的。

可以从<http://thinkpython2.com/code/Map.py>下载我的**HashMap**实现，但请记住并没有使用它的理由。如果需要映射，直接用**Python**字典即可。

## 21.5 术语表

**算法分析 (analysis of algorithms)**：通过对比运行时间以及/或者空间需求来对比算法的方法。

**机器模型 (machine model)**：用于描述算法的简化的计算机表示形式。

最坏情况（**worst case**）：让指定算法运行最慢（或者需要最多空间的）的输入。

首项（**leading term**）：在多项式中，指数最高的项。

交叉点（**crossover point**）：两个算法需要相同的运行时间或空间的问题规模。

增长量级（**order of growth**）：在算法分析时，如果我们认为一组函数的增长速度可以看作相等，则将这组函数称为同一个增长量级的。例如，所有线性增长的函数都属于同一个增长量级。

大O表示法（**Big-Oh notation**）：表达增长量级的方法。例如， $O(n)$ 表示所有线性增长的函数集合。

线性（**linear**）：运行时间和问题规模（至少对于大规模来说）成正比的算法。

平方量级（**quadratic**）：运行时间和 $n^2$ 成正比的算法，其中 $n$ 指的是问题规模。

搜索（**search**）：定位集合（如列表或字典）中某个元素或者判定它不在其中的问题。

散列表（**hashtable**）：一种表示键值对集合且搜索是常量级的数据结构。

---

[1] 但如果你在面试时被问到这个问题，我觉得更好的答案是：“给100万个数排序的最快算法应当是使用我用的语言提供的排序函数。它的性能应当对绝大多数应用都足够好了，但如果发现我的程序太慢，我会使用一个性能分析器去查看时间花在哪里。如果看起来更快的排序算法会带来明显的提升，那我会去寻找一个基数排序的良好实现。”

## 译后记

《像计算机科学家一样思考》这一系列书，早有耳闻，它可谓开创了程序设计入门书的一个新思路。授人以鱼，不若授人以渔；教人编程，不如引导人思考；教人语言细节，不若指明语言精要。而结合Python语言之后，得到的《像计算机科学家一样思考Python》这本书，则是在这个思路走到了一个极致的佳作。

我是工作之后才开始接触Python的。在那之前一直使用C/C++、Java、C#等传统风格的语言，再看到Python，不免有耳目一新之感。为何以往觉得晦涩难懂的程序设计理念，在Python中却表达得这么简洁易懂？为何以往需要绞尽脑汁才能拼出来的大段代码，在Python里却只需要几个简单调用即可？为何繁复的集合操作，在Python中却只需要一行列表理解循环语句就完成了？为何Python的文档那么容易找，还可以使用交互模式轻松尝试？每次使用Python编写程序之后，总会感慨，当初初学程序设计语言的时候，如果教的是Python该多好。相信所有学过C/C++之后再接触Python这类语言的人，都会有相同的感受吧。

那么是什么原因让C/C++几乎垄断了程序设计语言的教材呢？我觉得更多的是历史惯性。在计算机科学教育开始普及的20世纪70、80年代，C语言正在其鼎盛时期，几乎所有的人都在用C开发程序，操作系统、软件、游戏几乎都是用C甚至汇编开发的。硬件性能的限制，让那些更抽象、更高阶的语言，无法普及开来。因此教学自然也使用

它。久而久之形成了惯性，到了新世纪，程序设计的教学已经赶不上语言发展的潮流了。我们的程序越来越复杂，越来越像人脑，而教学的语言仍然在使用高级语言中最贴近机器的C。而C++、Java、C#，虽然相对于C更抽象高阶，但由于这些语言设计的初衷仍是以扩展C为主，所以不过是在这一惯性上多走了五十步而已。

本书正是扭转这种矛盾局面的一个有益的尝试。《像计算机科学家一样思考》是对程序设计教学模式的真谛的领悟，而使用Python这种简洁强大的高阶语言，也正是这种新思路最贴切的贯彻。授人以渔，自然应当用最好的渔具；引导人思考，当然也应使用更贴近人的思路而不是机器思路的语言。Python在高阶语言中，是一个从理念和实际综合考量后非常合适的候选。

在翻译过程中我发现，本书不但思路很贴切其教学主旨，从行文和用例来看也非常浅显易懂。全书讲了非常多的程序设计理念，在读过之后却会觉得那些理念都很自然，大概也是因为作者苦心安排，前后穿插，让读者能循序渐进地明白每个程序设计理念是因为什么而出现的原因吧。这种风格，再配合上精心编辑的示例，用于介绍任何程序设计语言，都是非常合适的。

如果将来我的孩子愿意学习程序设计，我愿意用这本书教他。

这一版，将语言升级到Python 3，从而更加贴近语言发展的趋势。作者对章节内容和示例练习也做出了重新组织和调整，使得阐述行文更加通畅。



尽管我已尽最大努力争取译文准确、完善，但仍然难免有疏漏之处，如发现问题，欢迎批评指正。电子邮箱[zhaopuming@gmail.com](mailto:zhaopuming@gmail.com)。

## 译者介绍

**赵普明** 毕业于清华大学计算机系，从事软件开发行业近10年。从2.3版本开始接触Python，工作中使用Python编写脚本程序，用于快速原型构建以及日志计算等日常作业；业余时，作为一个编程语言爱好者，对D、Kotlin、Lua、Clojure、Scala、Julia、Go等语言均有了了解，但至今仍为Python独特的风格、简洁的设计而惊叹。

## 作者介绍

Allen Downey是欧林工程学院（Olin College of Engineering）的计算机科学教授。他曾在韦尔斯利学院（Wellesley College）、科尔比学院（Colby College）和加州大学伯克利分校（U.C. Berkeley）任教。他从加州大学伯克利分校获得计算机科学博士学位，并从MIT获得硕士和学士学位。

## 封面介绍

本书封面的动物是卡罗来纳鹦鹉，也叫卡罗来纳长尾鹦鹉（学名 *Conuropsis carolinensis*）。这种鹦鹉分布于美国东南部，并且是一种栖息在墨西哥以北的大陆鹦鹉，它们最北曾一度到达纽约和大湖区，但主要分布在佛罗里达州到卡罗来纳州一带。

卡罗来纳鹦鹉主色是绿色，头部黄色，成熟时前额和两颊会出现一些橙红色的条纹。它的平均尺寸是31~33 cm。它叫声狂暴而巨大，并且在捕食过程中会喋喋不休。它居住在沼泽与河畔的树洞中。卡罗来纳鹦鹉是喜欢群居的生物，平时以小群体形式生活，在捕食时可以达到几百只。

不幸的是，这些捕食过程往往在农田的庄稼地里进行，农夫会射击它们，以免破坏庄稼。它们的群体特性让它们会集体救助受伤的鹦鹉，结果让农夫可以一次杀光整群鹦鹉。不但如此，它们的羽毛被用做妇女的帽饰，也有一些鹦鹉被作为宠物。这些因素组合起来，导致在19世纪晚期，卡罗来纳鹦鹉变得非常稀少，并且禽类疾病也加剧了它们的减少。到20世纪20年代，这个物种灭绝了。

今天，全世界的博物馆中保存了700多只卡罗来纳鹦鹉的标本。

很多O'Reilly的书封面上的动物都是濒危物种，它们全都对世界有重要意义。请访问[animals.oreilly.com](http://animals.oreilly.com)来了解如何帮助它们的信息。

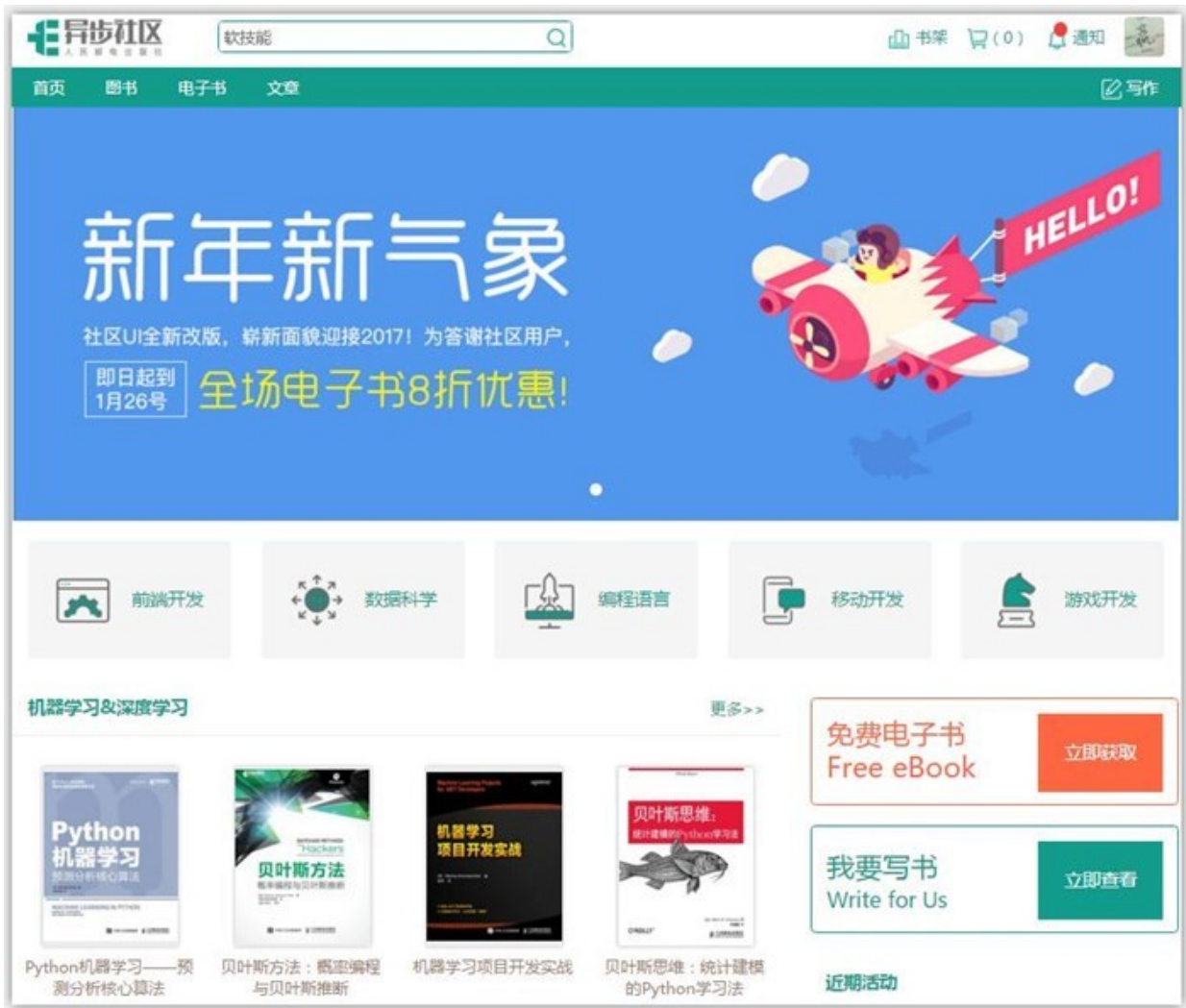
封面图片来自《约翰逊的自然历史》（*Johnson's Natural History*）。

# 欢迎来到异步社区！

## 异步社区的来历

异步社区([www.epubit.com.cn](http://www.epubit.com.cn))是人民邮电出版社旗下IT专业图书旗舰社区，于2015年8月上线运营。

异步社区依托于人民邮电出版社20余年的IT专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与POD按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



## 社区里都有什么？

### 购买图书

我们出版的图书涵盖主流IT技术，在编程语言、Web技术、数据科学等领域有众多经典畅销图书。社区现已上线图书1000余种，电子书400多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

### 下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

## 与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

## 灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。**100积分=1元**，购买图书时，在  里填入可使用的积分数值，即可扣减相应金额。

### 特别优惠

购买本电子书的读者专享**异步社区优惠券**。使用方法：注册成为社区用户，在下单购书时输入“**57AWG**”，然后点击“使用优惠码”，即可享受电子书8折优惠（本优惠券只可使用一次）。



## 纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

The screenshot displays the book page for "Wireshark网络分析的艺术" (The Art of Network Analysis with Wireshark). The page includes the book cover, author information (林沛满), and a detailed description. It features three purchase options: a discounted paper edition (¥31.50), an electronic edition (¥25.00), and a combined bundle (¥45.00). A bundle purchase of two paper editions is also shown for a total of ¥75.60. The page also includes a section for the author, LinPeiman, with a bio and a "Follow" button. There are tabs for "目录" (Table of Contents), "评论" (Reviews), "勘误" (Errata), and "出版信息" (Publication Information). A "精彩推荐" (Recommended) section at the bottom right suggests another book, "Nmap渗透测试指南" (Nmap Penetration Testing Guide).

**Wireshark网络分析的艺术**

作者：林沛满  
责编：傅道坤  
分类：计算机科学 > 安全与加密 > 网络安全

Wireshark是当前最流行的网络包分析工具。它上手简单，无需培训就可入门。很多棘手的网络问题遇到Wireshark都能迎刃而解。本书挑选的网络包来自真实场景，经典且接地气。讲解时采用了生活化的

5.6K 浏览 57 想读 7 推荐

下载PDF样章 配套文件下载

分享：

纸质 ~~¥45.00~~ ¥31.50 (7折) 电子 ¥25.00 电子 + 纸质 ¥45.00

购买

纸质 (纸质) + 纸质 (纸质) 总价：¥75.60 一起购买

目录 评论 9 勘误 1 出版信息

作者简介 专业书评 内容提要

**本书作者**

LinPeiman 上海 1.0K经验值

发私信 送积分 关注

《Wireshark网络分析就这么简单》即《Wireshark网络分析的艺术》作者

**兑换样书**

立即兑换 如何赚取积分

**电子书版本**

PDF Epub Mobi

**精彩推荐**

Nmap渗透测试指南 作者：南广明

## 社区里还可以做什么？

### 提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

### 写作

社区提供基于Markdown的写作环境，喜欢写作的您可以在这一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

## 会议活动早知道

您可以掌握IT圈的技术会议资讯，更有机会免费获赠大会门票。

## 加入异步

扫描任意二维码都能找到我们：



异步社区



微信订阅号



微信服务号



官方微博



QQ群: 436746675

社区网址: [www.epubit.com.cn](http://www.epubit.com.cn)

官方微信: 异步社区

官方微博: @人邮异步社区, @人民邮电出版社-信息技术分社

投稿&咨询: [contact@epubit.com.cn](mailto:contact@epubit.com.cn)